TextToSpeech Kit™ Programming Interface Version 2.0

Trillium Sound Research Inc.

March, 1995

Chapter 1

Introduction

1.1 Why Text-to-Speech?

One may ask why the TextToSpeech Kit is needed. The obvious answer is that it allows your computer to talk. Most feedback from computers occurs visually, with some tactile and simple auditory feedback. Giving your computer the ability to talk allows the user to interact with it in new ways, reinforcing visual information or providing an alternative to it.

Another compelling reason for text-to-speech is to provide a powerful new tool for developers of applications. Unrestricted text-to-speech conversion will be of special interest to programmers working with: integrated mail systems; teaching software; aids for the disabled; telephone access to databases; appointment systems requiring automated reminders to clients and patients; and any application where voice generated from arbitrary text provides a useful new modality. Imagine text editors which can speak the file you just typed in, or games with spoken messages, or educational software such as pronunciation dictionaries for languages or dialects. Almost every application can benefit from text-to-speech, and many as yet unimagined applications will arise once this capability is widely available.

Text-to-speech is also important at the system level of the computer. It is a natural extension to the Workspace Manager, moving along the same path towards a natural and intuitive interface with the computer.

Finally, text-to-speech allows computer access for the visually impaired. Presently, the blind are excluded from using most computers because of the technology's high dependence on visual feedback. When text-to-speech is properly coordinated with speech recognition, a completely new mode of interaction with computers will be created, benefitting not only the handicapped but all computer users in general.

1.2 What is the TextToSpeech Kit?

The TextToSpeech Kit is a collection of applications, programs, libraries, and documentation which allows the developer to incorporate text-to-speech capability into an application. The Developer TextToSpeech Kit contains the following items:

- **TextToSpeech Server.** The TextToSpeech Server and its associated databases do the actual conversion of text to speech, running as an independent task in the background. The databases include a large main dictionary, which provides accurate pronunciations for nearly 70,000 words. The Server and databases are installed in /LocalLibrary/TextToSpeech/system.

- **Client Libraries.** The TextToSpeech Object provides client access to the TextToSpeech Server, using Objective C conventions. The library libTextToSpeech.a is located in /usr/local/lib. Header files are installed in /LocalDeveloper/Headers/TextToSpeech.

- **PrEditor.** *PrEditor* (from **Pr**onunciation **Editor**) is an application that allows developers and end users to create and maintain supplementary pronouncing dictionaries, customized to their needs. *PrEditor* is located in /LocalApps/TextToSpeech.

- **SpeechManager**. *SpeechManager* provides the ability to customize the execution characteristics of the TextToSpeech Server for any given system. Your system administrator uses this application to tune the Server if speech output "chatters" or suffers from frequent interruptions. It is installed in /LocalApps/TextToSpeech.

- **BigMouth.** *BigMouth* is an application which provides a text-to-speech service to other applications. It is installed in /LocalApps/TextTo-Speech.

- **ServerTest.** *ServerTes*t is an application which makes all the Server's methods available for testing and experimentation through a simple graphical interface. It is located in /LocalApps/TextToSpeech.

- **WhosOnFirst**. An application that generates voice alerts for system activity, including application launches and remote logins. It is included in /LocalApps/TextToSpeech.

- **say**. A command line interface to the TextToSpeech system, which can be used from a terminal or in shell scripts. *say* is installed in /user/local/bin.

- **Documentation.** Complete developer documentation is provided, with reference material and tutorials. Laser-printable copies are provided, as well as fully indexed RTF

documentation for Digital Librarian. The documentation is located in /LocalLibrary/TextToSpeech/documentation.

- **Example Code.** Complete source code for *BigMouth* and *ServerTest* is provided. These should provide useful examples to developers when they incorporate text-to-speech capabilities into their own applications. The source code is installed in /LocalLibrary/TextToSpeech/exampleCode.

- **Fonts**. Three specially developed fonts, *Lexical*, *Phonetica*, and *Trillium-Phonetic* are included to enable phonetic editing of pronunciations with *PrEditor*. These fonts are installed in /LocalLibrary/Fonts.

- **Device Driver.** A device driver for the Turtle Beach MultiSound DSP card is provided for Intel computers.

The TextToSpeech User Kit contains the following items:

- TextToSpeech Server
- PrEditor
- SpeechManager
- BigMouth
- WhosOnFirst
- say
- Fonts
- Device Driver

The User Kit is intended to provide the basic functionality needed to run applications developed with the TextToSpeech Developer Kit. The end user is thus spared the expense of purchasing unnecessary Kit components.

1.3 Support for Intel Platforms

In addition to supporting NeXT Computers, the TextToSpeech Kit will now also work on Intel platforms running NEXTSTEP if equipped with the Turtle Beach MultiSound DSP/Sound card. An enhanced Music Kit driver for this card is bundled with the TextToSpeech Kit, and must be installed before running the TextToSpeech software. This driver is a superset of the stock Music Kit driver provided by CCRMA, so all Music Kit functionality is retained. Note, however, that the standard Music Kit driver does not support the TextToSpeech Kit. Trillium Sound Research and CCRMA are cooperating to provide a unified Music Kit driver in the future.

The TextToSpeech Kit now supports a “software’’ speech synthesizer that doesn’t use the DSP.

This synthesizer runs on the host CPU and outputs sound samples to file. It is far too slow to operate in real-time, but it may be useful for Intel machines which aren't equipped with a DSP board but do have the capability to play sound files through a simple (i.e. non-DSP) sound card.

1.4 Organization of this Manual

Chapter 2 gives an overview of the concepts needed to understand most parts of the TextToSpeech Kit. Chapter 3 provides several tutorial examples of how to program with the Kit, with detailed step-by-step instructions. Chapters 4 and 5 are detailed references for the TextToSpeech class. They should be referred to whenever a comprehensive technical description of any of the class methods or instance variables is needed.

Chapter 2

Concepts

2.1 Client-Server Structure

The TextToSpeech Kit is implemented using a client-server structure. The *TTS_Server* is an independent Mach task which operates in the background; it is the program that performs the actual conversion of text to speech. Client applications communicate with the Server so it will speak the desired text, and to control how the text is spoken.

The client-server structure has several advantages over an implementation which uses a traditional object library:

- The application avoids linking in a large amount of code, keeping the application to a reasonable size.

- There is no needless duplication of code whenever two or more applications make use of the text-to-speech facility.

- **The Server can be updated at any time without affecting the applications which use it. This means that improvements to the text- to-speech system can be made transparently without recompiling applications.**

Applications do not communicate with the Server directly, but indirectly using the Objective C methods of the *TextToSpeech Object*. When the application instantiates a TextToSpeech Object, it creates a connection to the ServerÐthe application is said to occupy a "client slot". When the Object is freed, the connection is severed, and the client slot can then be used by another application.

The Server allows up to 50 clients. This does not mean that the Server will synthesize 50 voices simultaneously, but that 50 connections can be made to the server at the same time. See Figure 2.1. Each connection can be configured differently, so, in effect, 50 different kinds of voices can exist at once. This also means that a single application can create several connections to the

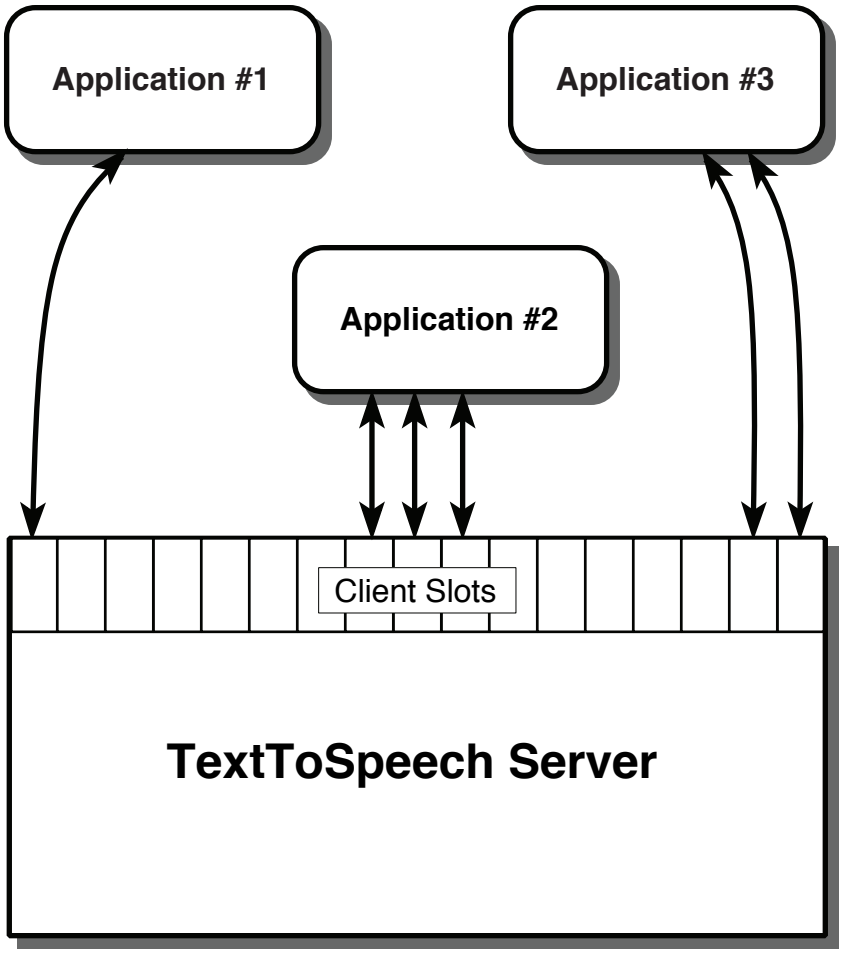

Figure 2.1: Applications accessing the TextToSpeech Server

Server by instantiating as many TextToSpeech Objects as needed. This may be useful when an application needs a different sounding voice for each of several functions.

Figure 2.1: TextToSpeech Server allows up to 50 client connections. Client applications may use several connections simultaneously.

The operation of the TextToSpeech Server is transparent to both the programmer and user. The Server is started up automatically when needed, and terminates itself when it detects that no clients are connected. Control over any attribute of the Server is accomplished by sending the appropriate message to the TextToSpeech Object. Each method of the Object performs the necessary intertask communication with the Server.

2.2 Operation of the Server

The basic operation of the Server is outlined in Figure 2.2. The Server has three main Modules, each of which performs a specific task.

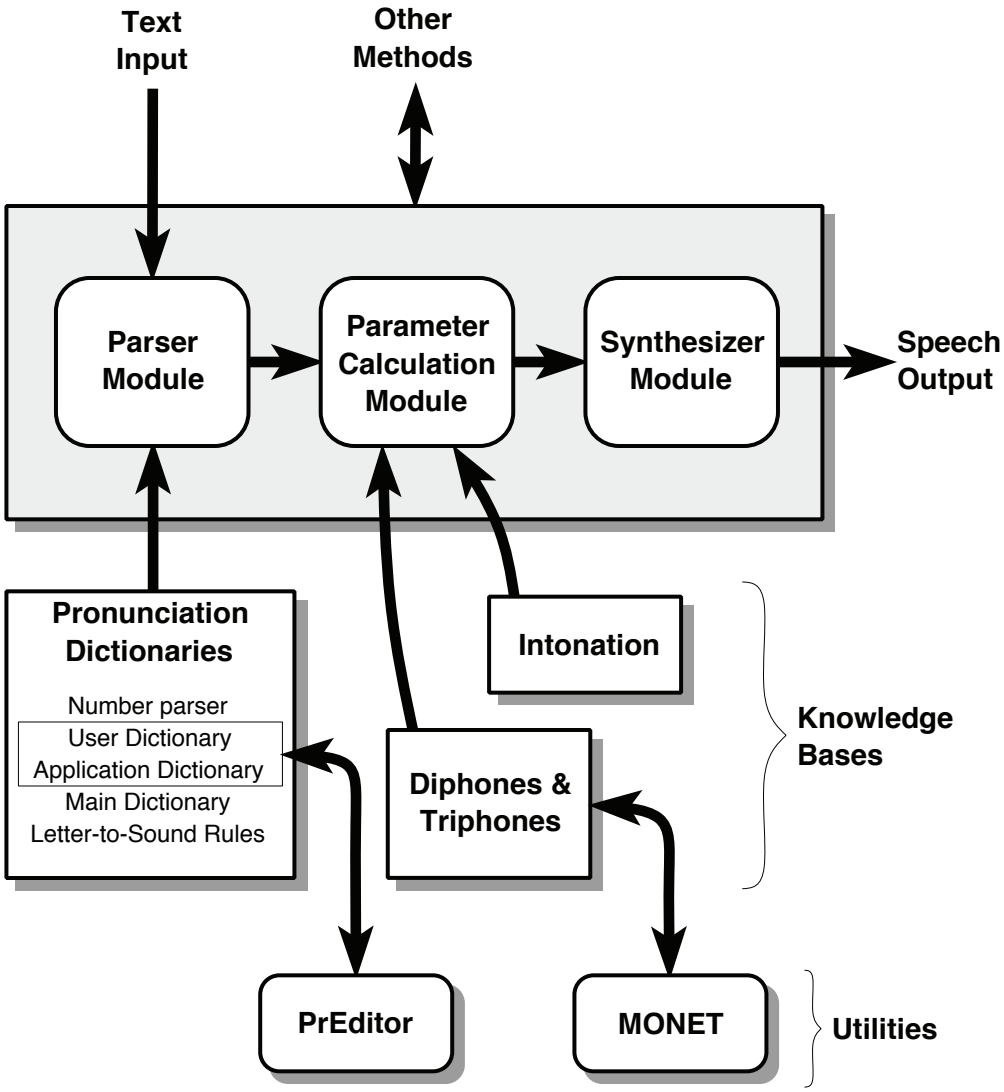

Figure 2.2: Basic form of the TextToSpeech Server

2.2.1 Parser Module

The Parser Module takes the text supplied by the client application (using the **speakText:** or **speakStream:** methods) and converts it into an equivalent phonetic representation. The input text is parsed, where possible, into sentences and *tone groups*. This subdivision is done primarily by examining the punctuation. Each word or number or symbol within a tone group is converted to a phoneme string which indicates how the word is to be pronounced. The pronunciation is retrieved from one of five pronunciation knowledge bases.

The Parser must also deal with text entered in any of the special text modes. For example, a word may be marked in *letter mode*, which means the word is to spelled out a letter at a time, or in *emphasis mode*, which means the word is to receive special emphasis by lengthening it and altering its pitch. The Parser marks the phonetic representation appropriately in these cases.

Note that the order in which the pronunciation knowledge bases are searched is under program control, and customized pronunciations for particular words can be entered into the User and Application Dictionaries using the *PrEditor* utility.

2.2.2 Parameter Calculation Module

The Parameter Calculation Module takes the phonetic representation produced by the Parser Module and converts it into parameter tracks suitable for the Speech Synthesizer. The phonemes are converted into diphones or triphones, which are then used to create control data for each parameter of the Synthesizer. This module also creates an appropriate intonation contour for the utterance, as well as imposing speech rhythm.

The user has some control over how the Parameter Calculation Module operates. For example, the extent and type of intonation can be controlled, as can speaking rate and pitch level.

2.2.3 Synthesizer Module

The Synthesizer Module takes parameter control data from the previous module and uses it to synthesize the speech in real-time. The Synthesizer is a program which runs on the DSP chip. It physically models the acoustics of the vocal and nasal tracts using articulatory control parameters. The user can control the synthesizer in various ways, including altering the length of the vocal tract and the breathiness of the glottal source.

The output of the DSP chip is a digital signal which is converted by the Digital-to-Analog Converter into an audio signal. This signal can be heard on the internal loudspeaker (NeXT computers only) or, if connected, on an external audio system. The DSP and sound hardware is used by the TextToSpeech Kit only when an utterance is being synthesized. At all other times, the hardware is free for system beeps and other audio output. Note that the output of the DSP synthesizer can be directed to a sound file instead of real-time output if desired.

The synthesizer can also be run on the host CPU instead of the DSP. This “software synthesizer’’ is very slow, so real-time output is not possible, and output can only be sent to a sound file. This option is useful for Intel platforms which aren’t equipped with a DSP board, but can play sound files through a simple sound card.

Figure 2.2: The basic components of the TextToSpeech Server are the Parser Module, Parameter Calculation Module, and Synthesizer Module. Knowledge Bases which are accessed by the Server are maintained with the Utilities *PrEditor* and *MONET*.

2.3 Text Input

2.3.1 Standard English Text

The TextToSpeech Kit is designed to handle standard English text as its input. Standard punctuation and capitalization conventions are assumed. Clear guidelines can be found in *The Elements of Grammar, by Margaret Shertzer (New York: MacMillan Publishing Company, 1986).*

The system attempts to speak the text as a person would. Punctuation is not pronounced, but is used as guide to pronounce the text it marks. For example, a period that marks the end of sentence is not pronounced, but does indicate that a pause occurs before proceeding to the next sentence. A question mark behaves similarly, but also indicates that the pitch should rise at the end of the sentence. Of course, if you wish to pronounce the punctuation marks, you will have to use a special input mode (see below) or pre-parse the text.

The following list summarizes the punctuation conventions used by the TextToSpeech Kit:

1. Only characters in the standard 128-character ASCII set are recognized. All others are converted to blanks.

2. Only *"printing characters" (*`040` to `0176`) and the escape character (which is user-definable) of the ASCII set are used. All other "control characters" are converted to blanks.

3. Periods, question marks, and exclamation points are usually considered punctuation which ends a sentence, and a long pause is inserted after each. These punctuation marks may also be used in other contexts where they do not mark the end of a sentence (see below), and thus will not cause the insertion of a pause.

4. Commas, colons, and semicolons are usually considered punctuation which separate phrases within a sentence. A short pause is inserted after each comma, and a somewhat longer pause is inserted after the colon and semicolon. Commas and colons may appear in other contexts where they do not mark phrase endings (see below), and thus won't cause the insertion of a pause.

5. Words containing an apostrophe are handled appropriately. The contraction or possessive is pronounced, but the apostrophe itself is not.

6. Single and double quotation marks are not pronounced.

7. Dashes (`--` or `---`) are not pronounced, but do insert a pause.

8. Parentheses and square brackets are not pronounced, but will cause pauses to be inserted if they delimit parenthetical words or phrases.

9. Ellipsis dots (3 periods with optional spaces) are not pronounced, and do not insert a pause.

10. Hyphenated words are treated as two separate words unless the hyphenation is at the end of the line, in which case the hyphenated parts are joined together. The hyphen is not pronounced in either case.

11. Forward slashes are not pronounced. Constructions such as "high/low", or "he/she" are treated as two separate words.

12. The following punctuation marks are treated as symbols if they appear unconnected to any other characters, and are explicitly pronounced:

    (a) `&` "and"
    (b) `+` "plus"
    (c) `<` "is less than"
    (d) `>` "is greater than"
    (e) `=` "equals"
    (f) `-` "minus"
    (g) `@` "at"

13. The following punctuation can be used with numbers, causing them to be pronounced in the appropriate way:

    (a) The period in decimal numbers.
    (b) The comma in numbers such as `1,456`.
    (c) The colon in clock times such as `10:23`.
    (d) The hyphen to indicate negative quantities, and in telephone numbers.
    (e) The plus sign to indicate positive quantities.
    (f) Parentheses in telephone numbers such as `(345)555-1234`.
    (g) The forward slash in fractions such as `4/3`.
    (h) The dollar sign.
    (i) The percent sign.

14. The following punctuation marks are always converted to blanks, unless in letter mode:

    `#  *  \  ^  _  |  ~  {  }  (!)  (?)`

    The punctuation `(!)` and `(?)` is used occasionally to express irony or sarcasm.

Some characters perform more than one function, depending upon context. For instance, the period can be used to mark the end of a sentence, as part of ellipsis dots, to indicate a decimal number, or to delimit an abbreviation (see below). The hyphen can be used to split words over two lines, to hyphenate two words, as part of a dash, to indicate a negative quantity, as part of a telephone number, or to act as the minus sign symbol. The Server's parser is sophisticated enough to recognize all these cases.

Common unambiguous abbreviations are always expanded to their full equivalent, so a phrase like "Prof. John Smith, Sr." is pronounced as "Professor John Smith, Senior". Single letters followed by a period are pronounced as in letter mode. For example, the phrase "David J. Brown" is pronounced as "David `jay' Brown". Note that many abbreviations are not recognized by the system because they can expand to two or more different words. In these cases, the period is treated as the end of a sentence, and the abbreviation is pronounced without expansion.

When every letter of a word is capitalized, the word is pronounced one letter at a time as, for example, in the sentence "He gave the CIA's files to the NBC network chief." The system will also pronounce this correctly if the capitalized words are punctuated with periods, as in "C.I.A." and "N.B.C." (Current practice is to omit the periods in such cases.) Some capitalized words, such as NATO or UNICEF, are not pronounced one letter at a time, but with a word-like pronunciation instead.

2.3.2 Parsing of Numbers

Any contiguous string which contains a digit is handled by the number parser. The number parser will pronounce the following cases correctly:

Cardinal numbers. Numbers up to 10^{63} (vigintillion) are pronounced with the proper triad name; numbers longer than this are pronounced one numeral at a time. For example, `1547630` is pronounced "one million <pause> five hundred and forty-seven thousand <pause> six hundred and thirty." Cardinal numbers can have commas every three digits. For example, `23,567` is pronounced "twenty-three thousand <pause> five hundred and sixty-seven."

Ordinal numbers. Ordinal numbers up to 10^{63} are pronounced correctly, provided the proper suffix (-st, -nd, or -th) is given. E.g. `101st` is pronounced "one hundred and first."

Positive and negative numbers. A `+` or `-` sign can be placed before all numbers, except before telephone numbers, clock times, or years. For example, `-34` is pronounced "negative thirty-four."

Decimal numbers. All numbers with a decimal point are pronounced. For example, `+34.234`

is pronounced "positive thirty-four point two three four."

Percent. If a `%` sign is placed after the number, the word "percent" is also pronounced. E.g. `2.45%` is pronounced "two point four five percent."

Simple fractions. Numbers in the form *integer/integer* are pronounced as fractions. Each integer must not contain commas or decimals. Any `+` or `-` sign must precede the first integer, and any `%` sign must follow the last integer. E.g. `-3/4%` is pronounced "negative three quarters percent."

Dollars and cents. Dollars and cents are pronounced correctly if the `$` sign is placed before the number. An optional `+` or `-` sign may precede the dollar sign. For example, `-$2.01` is pronounced "negative two dollars and one cent."

Telephone numbers. The number parser recognizes the following types of North American telephone numbers and pronounces them appropriately:

- **7 digit code. E.g.** `555-2345` is pronounced "five five five <pause> two three four five."
- **10 digit code.** E.g. `203-555-2345` is pronounced "two zero three <pause> five five five <pause> two three four five."
- **11 digit code.** E.g. `1-800-555-2345` is pronounced "one eight hundred <pause> five five five <pause> two three four five."

Clock times. The number parser correctly pronounces both normal and military time. For example, `9:31` is pronounced "nine thirty-one," and `08:00` is pronounced "oh eight hundred." Seconds are also recognized. For example, `10:23:14` is pronounced "ten twenty-three and 14 seconds." Non-military times on the hour have o'clock appended. E.g. `9:00` is pronounced "nine o'clock," but `14:00` is pronounced "fourteen hundred."

Years. Integers from 1000 to 1999 are pronounced as two pairs. For example, `1906` is pronounced "nineteen oh six," not "one thousand nine hundred and six." If the second pronunciation is preferred, a comma should be inserted after the first digit, as in `1,906`.

Mixed characters. If a word contains digits and other non-numeric characters mixed together in a format not described above, it will be pronounced one character at a time.

2.3.3 Special Input Modes

Special Input Modes allow the programmer to have text pronounced in special ways. *Letter*

mode indicates that the marked text (including blanks) is to be pronounced one letter at a time. *Emphasis mode* is used to emphasize particular words or phrases. *Silence mode* allows the programmer to insert arbitrary lengths of silence into the text.

The programmer must embed an escape code to indicate the beginning or end of a mode. The escape codes are always three characters long, and have the following format:

ESC *modeSymbol modeMarker*

The **ESC** character (hexadecimal `1B`) indicates the beginning of an embedded escape code. It can be redefined by the programmer to any ASCII value (except NULL), if desired.

The escape character must always be followed by 2 characters. The *modeSymbol* is a character which indicates which mode is being invoked. The available modes and their *modeSymbols* are:

Mode	**Symbol**
Letter	l or L
Emphasis	e or E
Silence	s or S

The *modeMarker* is a single character which indicates the beginning or end of a mode. The letters **b** or **B** indicate the beginning of the mode, while the letters **e** or **E** indicate the end of the mode.

Escape codes should never be embedded in the middle of a word. Escape codes can be nested, if desired. An error code is returned if the nesting is improperly done.

Letter mode causes one or more words or symbols to be spelled out one character at a time, rather than pronounced as a word. For example, if you wanted the words “fare” and “fair” to be pronounced a letter at a time in the following sentence, the escape codes would be embedded as shown:

```
He meant the word ESC-LBfairESC-LE, not ESC-LBfareESC-LE.
```

Any amount of text can be put into letter mode. Once in letter mode you must leave it explicitly using the ESC-le code. The letter mode escape codes should never be placed in the middle of a word, since a word must be either solely spelled or spoken.

Letter mode is useful for spelling out unpronounceable strings. Data files or program source files, for example, often contain text which is not standard English. Since strings like `*/` or `1C4A3F` are obviously unpronounceable as words, they should be sent to the TextToSpeech Object in letter mode. Letter mode is also useful when a number is to be pronounced one digit at a time, rather than as would be normally done. For example, `ESC-lb547ESC-le`

would be pronounced as “five four seven” in letter mode, but “five hundred and forty-seven” otherwise.

Emphasis mode is used to emphasize particular words or phrases in an utterance. The emphasized word or phrase receives special stress, usually by making it longer, louder, and higher in pitch. The TextToSpeech system creates the emphasis automatically, and connects the emphasized parts naturally with the rest of the words of the utterance. Note that the emphasis escape code cannot be placed in the middle of a word because only the whole word can be emphasized; partial emphasis of a word is not allowed.

Emphasis is often implied in text when a word is italicized or underlined. It is usually applied to a word which normally does not receive emphasis in a sentence---by placing the stress on that word the meaning of the utterance may be altered. For example, placing emphasis on the word “was” changes the normal declarative meaning of the following sentence, and implies that what was once true is no longer so:

```
It ESC-eb was ESC-ee a nice place.
```

Silence mode may be used by the programmer to insert silence between words. The amount of silence is specified in seconds or fractions of seconds with a decimal number. The specified length of silence will be rounded to the nearest 10th of a second, since that is the resolution of control. One must remember that silence is automatically inserted between words wherever a pause will naturally occur, such as after a period or comma. Silence mode should not be used to replace such naturally occurring pauses, but rather to specify arbitrary lengths of silence between words or utterances. If silence mode is used where there would normally be a pause, then that pause is discarded and the length of the pause becomes that which is specified with the embedded escape code. Silence mode cannot be used in the middle of a word; pauses are allowed between words only.

Silence is specified by embedding the escape code `ESC-sb` *time* between words. For example the text:

```
You said what?  ESC-sb 3.0 What?
```

has a 3 second silence inserted between the repetition of the words “what”. Normally there would be a short pause after the question mark, but in this case the pause is arbitrarily specified by the programmer.

The `ESC-se` escape code is not necessary after the time, but can be inserted if desired to be consistent with letter and emphasis modes. Remember that the time values assigned using the silence mode are ASCII character representations of an integer or real number, not the integer or real data type itself. This is because the time value must be embedded within a character string.

person might actually speak the file. This filtered text would then be sent to the TextToSpeech Object to be heard. The filter, in this case, would be quite complex, so we cannot describe it fully here. An obvious place to start is with comments. Each `/*` would be converted to "begin comment", and each `*/` to "end comment", and the comment itself would probably be passed straight through without any changes. Punctuation might have to be inserted by the filter so that there are appropriate pauses.

The **speakStream:** method of the TextToSpeech Object is designed so that incorporating filters into an application is easy. A memory stream is ideal for handling the input and output of a filter, since it allocates memory as needed. This is important, since a filter may expand, by a large factor, symbols and cryptic text into full English descriptions. A chain of filters could be created, the output of one feeding the input of the next. The output of the last filter would, of course, be fed into the **speakStream:** method of the Object. See Figure 2.3.

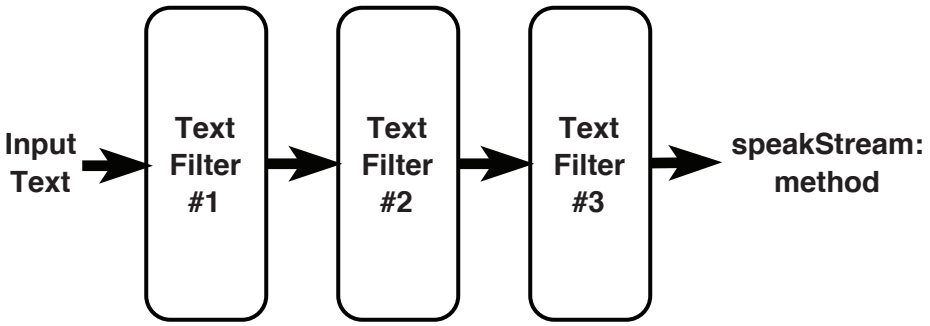

Figure 2.3: A chain of text filters can be devised to handle particular kinds of input text.

2.3.4 When is Pre-parsing Necessary?

Pre-parsing is necessary whenever the text to be spoken it is not standard English. For example, program source code, documents containing technical jargon, UNIX commands, etc., do not follow the conventions of standard English, and thus must be pre-processed appropriately to produce the desired speech. The nature of this pre-parsing depends on the type of text, and often upon several different contexts within that text. Consequently, no universal parser exists which can handle every possible situation. The TextToSpeech Kit provides the most basic parsing possible given that it does not know the context of the text it is converting. It is the responsibility of the application developer to pre-parse the text when necessary, since only he or she will know what sort of text is to be handled.

As an example, consider how one might "speak" a C program file. It obviously is not standard English, so the conventions described above are not appropriate to render it into speech. The

programmer must write a special “filter” to convert the file into a text which represents how a

2.3.5 Languages Other Than English

The TextToSpeech Kit is expressly designed to render the English language in the best way possible. The main pronouncing dictionary, number parser, and letter-to-sound rules are all English-based, as are the phonemes, intonation, and rhythm patterns. To speak another language properly, all these elements would have to be redesigned and reimplemented.

A limited approximation of another language is possible if the user supplies the pronunciations of foreign words in the Application or User pronunciation dictionaries. The 40 phonemes of English used in the system match many of the phonemes of other languages, although the match is not exact in some cases. Some phonemes found in other languages are not included in the present system, so not all foreign words can be pronounced properly. However, a fairly good approximation can be found in most instances.

2.4 Controlling the Speech in Real-Time

Once the text to be spoken has been entered into the system using the **speakText:** or **speakStream:** methods, control returns immediately to the calling routine. The speech is synthesized in the background and will normally continue until the entire utterance has been spoken. Since the speech is performed by a separate Mach task, the client application can do other things as the speech is being produced. Several new utterances can be queued up to speak while old utterances are being synthesized, from several client applications if necessary.

The speech can be paused at any time by sending a **pauseImmediately** message to the TextToSpeech Object, and resumed by sending a **continue** message. There is a slight delay when pausing since, by virtue of the NeXT’s architecture, the sound is pipelined as it is synthesized. Once paused, more utterances can be queued up if desired. The speech can also be paused at the end of the current utterance by using the **pauseAfterCurrentUtterance** method. When a **continue** message is sent, any remaining utterances will then be synthesized.

If you wish to stop all speech output without the possibility of resuming, you should send an **eraseAllSound** message to the TextToSpeech Object. This message can be sent in either the paused or unpaused state. All queued up utterances are erased, which allows immediate entry of new utterances. The **eraseCurrentUtterance** erases only the currently speaking utterance, which means queued utterances will then speak immediately. If this message is sent while in a paused state, the pause which is associated with the current utterance is also erased, so any subsequent utterances will be heard immediately.

2.5 Controlling Speech Quality

2.5.1 The Articulatory Speech Synthesizer

The TextToSpeech Kit uses a new and innovative speech synthesizer that directly simulates the acoustic properties of the human vocal and nasal tracts. The chief advantage of this *articulatory synthesis* technique is the naturalness of the resulting voice quality. Since the synthesizer directly models the human vocal apparatus, seven or eight formants (resonances) are always naturally produced at their proper frequencies. In contrast, "source-filter" synthesizers have three or four variable-frequency formants, and the higher formants, if any, are usually fixed in frequency. For this reason, the older technology will always produce speech which sounds "robotic" and unnatural. Other problems abound. Nasal sounds are notoriously poor using traditional synthesizers, since they are usually simulated (with varying degrees of success) with notch filters and other kludges. The articulatory synthesizer, on the other hand, physically models the nasal tract, so it is straightforward to produce accurate and natural-sounding nasals.

Unlike "Klatt-style" speech synthesizers, it is very easy to create a variety of voice types with the articulatory synthesizer: one simply specifies values for the parameters which describe the physical characteristics of the person producing the speech. For example, to synthesize the voice of a large female with a rather low pitch who has smoked for 30 years, one might specify a vocal
tract length of 16 centimeters, a median pitch of middle C, and a large amount of "breathiness". A child's voice might be specified with a vocal tract length of 13 centimeters, a median pitch of E above middle C, and almost no breathiness. Of course, an almost infinite number of combinations are possible, each of which can be prescribed intuitively by the user or programmer, and easily produced by the synthesizer.

The difference between male, female, and child voices is due mainly to two factors: vocal tract length and median pitch. These two parameters are not directly set in the TextToSpeech Kit, but are adjusted or "offset" from baseline values arbitrarily determined for each voice type. These baseline values are summarized here:

	Male	**Female**	**Large Child**	**Small Child**	**Baby**
length	17.5	15.0	12.5	10.0	7.5
pitch	±12.0	0.0	2.5	5.0	7.5

The vocal tract length is given in centimeters, and the median pitch is given in semitones, with 0 equal to middle C. The vocal tract length tends to vary with the height of the person, so, as a rule, men have longer vocal tracts than women, who have longer vocal tracts than children. The pitch of the voice depends primarily upon the size of the vocal cords. Males who have reached puberty have considerably longer vocal cords than women or children, so their voices are roughly an octave (12 semitones) lower. Of course, the pitch of the voice rises and falls over a

large range when speaking or singing, so the values above are “median” or average pitch levels.

The user or programmer specifies the voice type with the **setVoiceType:** method. This sets the median pitch and vocal tract length to the values listed above. If desired, the user “offsets” from the baseline values using the **setPitchOffset:** and **setVocalTractLengthOffset:** methods. For example, if the user sets the voice type to female, the pitch offset to 1.5, and the vocal tract length offset to ±0.5, the resulting median pitch will be 1.5, and the vocal tract length 14.5 centimeters. These offsets are retained when the voice type changes. For example, if the female voice type above is changed to a male voice, but the offsets are not altered, the resulting median pitch will be ±10.5, and the vocal tract length 17.0 centimeters. In a sense, these two voices “correspond” to each other, since they are both slightly shorter, and pitched somewhat higher, than the “average” male or female voice.

2.5.2 Limitations Due to Hardware Speed

The articulatory speech synthesizer is extremely compute intensive, since it physically models the propagation of sound pressure waves through the vocal and nasal tracts, sampling both in time and space. It uses a variable internal sample rate to simulate vocal tracts of different length (note, however, that the output sample rate is at one of the standard rates, either 22050 or 44100 Hz.). Shorter vocal tract lengths use higher sample rates, which means that more computation is required. Although the DSP is a very fast dedicated coprocessor, it is not fast enough to synthesize speech in real time with vocal tract lengths shorter than 15.9 centimeters on NeXT computers, and shorter than 16.6 centimeters on platforms equipped with the Turtle Beach MultiSound DSP card. If the user attempts to synthesize speech in real time below these limits, an error code is returned and the synthesis is not done.

Of course, the user always has the option of synthesizing to file, and playing back that file at a later time. Vocal tract lengths as short as 7.95 centimeters can be used with the DSP synthesizer. There is one restriction, though. If the vocal tract length is set below 15.9 centimeters, only the 44100 Hz. output sample rate can be used. If the user attempts to synthesize to file using the lower rate, an error code is returned, but the file is generated at the higher sample rate.

The user also has the option of using the “software synthesizer”. This runs on the main CPU (i.e. it doesn’t use the DSP), and therefore is extremely slow. It can only be used to synthesize to file, but it has the advantage that the vocal tract can be any length, and there is no restriction on which output sample rate can be used. The speech quality is slightly higher, since a variable-shaped glottal pulse is used, and control parameter information is interpolated at the sample rate. The software synthesizer may be useful on machines that aren’t equipped with a DSP card, but expect long waits when producing sound files.

2.5.3 Methods to Control Speech Quality

The TextToSpeech Kit provides comprehensive control over several aspects of the speech quality. The messages which control the speech quality can be sent at any time, but their effect will not be heard until the end of the sentence currently being spoken. This is so because the control parameters for the speech are calculated one sentence at a time. Detailed descriptions of each of the speech quality methods can be found in Chapter 5.

As noted above, the programmer uses the **setVoiceType:**, **setPitchOffset:**, and **setVocalTractLength:** methods to to set the median pitch and vocal tract length of the voice. These methods tend to have the largest effect upon speech quality. The voice type can be set to male, female, large child, small child, or baby. Median pitch can be raised or lowered 12 semitones (one octave). Of course, if intonation is enabled, the pitch of the voice will vary around the center pitch. The vocal tract can be lengthened or shortened as much as 3 centimeters. These offsets are retained whenever the voice type is changed. Note that shorter vocal tract lengths cannot be produced in real time, and that there are some restrictions when using the DSP to synthesize to file. See Section 2.5.2 above, and the method descriptions in Chapter 5.

The **setBreathiness:** method determines how much "noise" is introduced into the glottal source (the vocal cords). A value of 0 indicates that no breathiness is to be added, and the resulting voice quality sounds "pure". As more breathiness is added (to a maximum of 10), the voice becomes "rougher" in quality, or more hoarse.

The volume of the synthesized voice can be controlled independently of the keyboard volume controls using the **setVolume:** method. The volume has a range of 0 to 60 decibels, although the volume should normally never be set to less than about 48 dB.

The stereo balance of the synthesized voice can set using the **setBalance:** method. The sound will appear at the left speaker if set to ±1.0, at the right speaker when set to +1.0, and midway between the two speakers when set to 0.0. Of course, the voice can be placed anywhere between these points in the stereo field by using the appropriate value within the range ±1.0 to +1.0. Note that a change of balance will only be heard if the output has been set to 2 channels. The method **setNumberChannels:** allows the user to hear either mono or stereo output. A mono setting is useful when synthesizing to file, since the resulting sound file will be half as large as that done with 2 channels.

Changing the speaking rate of the voice is accomplished by sending a **setSpeed:** message to the TextToSpeech Object. Its argument is a speed factor: when set to 2.0, the speed of the speech is twice the normal rate; when set to 0.5, the speed is one half the normal rate. The allowable range is from 0.2 to 2.0.

The type and extent of speech intonation can be controlled with the **setIntonation:** method by

setting particular bits of its argument. If "micro-intonation" is specified, variations in pitch due to air pressure differences (associated with events such as stops and plosives) and larynx height changes will be added to the speech. "Macro-intonation" is the variation of pitch due to syllable stress, phrase type, and other supra-segmental effects. It is the parameter with the greatest effect over the pitch, and helps to give speech its meaning. "Declination" is the gradual drop in pitch that naturally occurs as the lungs run out of air. If the "randomize" bit is set, slight random variations are added to the macro-intonation. This helps the speech sound less monotonous.

2.6 Customizing Pronunciations

Although the Main Dictionary provides accurate pronunciations for an enormous number of words, it may not contain pronunciations for some specialized terms, newly coined words, misspelled words, surnames, or place names. The utility *PrEditor* is designed to allow the user or application developer to create User and Application Dictionaries that contain customized pronunciations. These dictionaries are usually placed before the Main Dictionary in the search order, so that any pronunciations they contain will override or supplement the pronunciations found in the Main Dictionary.

The default dictionary search order is:

1. Number Parser algorithm,
2. User Dictionary,
3. Application Dictionary,
4. Main Dictionary, and
5. Letter-to-Sound algorithm.

This order can be changed with the **setDictionaryOrder:** method. However, unless there is a good reason to change the search order, the default is usually preferred since it allows User and Application Dictionaries to override the Main Dictionary. The Number Parser should be kept first in the search order so that it immediately catches any numbers; the other dictionaries cannot handle numbers very effectively. The Letter-to-Sound algorithm is usually kept last in the search order since the pronunciations it provides, though usually acceptable, are not as as good as the hand-edited pronunciations found in the other dictionaries. It can be thought of as a "catch-all" which provides pronunciations when none of the other dictionaries can do so.

The User Dictionary is created by a user and stored on a per-account basis. The path to this dictionary is stored in the user's default database. When an application instantiates a TextToSpeech Object, this path is used as a pointer to the User Dictionary. The path to the User Dictionary is usually never explicitly set by an application, but the method **setUserDictPath:** is provided for the rare case where the programmer needs control of this.

The Application Dictionary is created by the application developer when it is known that the application will use specialized words not present in the Main Dictionary. The developer is responsible for creating and maintaining the dictionary (using *PrEditor*), and for linking the dictionary to the application using the **setAppDictPath:** method. The Application Dictionary is usually kept in the application's file package.

PrEditor is supplied with both the User and Developer TextToSpeech Kits so that both users and developers can create customized pronunciations. It is located in /LocalApps/TextToSpeech. *PrEditor* contains extensive online documentation which can be printed out.

Chapter 3

Programming Tutorial

3.1 General Programming Procedures

An application communicates with the TextToSpeech Server using various methods of the *TextToSpeech Object*. The TextToSpeech Object is a subclass of the superclass called *Object*, and like any other class, can itself be subclassed.

The operation of the TextToSpeech Server is normally completely transparent to both the programmer and user. It is started automatically when needed, and terminates itself whenever it detects that no clients are connected to it. The client application never communicates with the Server directly, but accesses its services using the methods of the TextToSpeech Object. These methods are described in detail in Chapter 5.

In order for an application to instantiate and communicate with the TextToSpeech Object, it must be compiled with the TextToSpeech client library. This library is installed in:

```
/usr/local/lib/libTextToSpeech.a.
```

You can specify this library in Project Builder simply by adding lib`TextToSpeech.a` to the "Libraries" suitcase in the Files part of Project Builder.

The header file TextToSpeech.h must be included in every Objective C file which refers to the TextToSpeech class. This is usually done with the directive:

```
#import <TextToSpeech/TextToSpeech.h>
```

This file specifies the Objective C interface to the TextToSpeech class. The header file, and file it includes, also contain definitions and datatypes used by the TextToSpeech Object. The header files are installed in the subdirectory:

```
/LocalDeveloper/Headers/TextToSpeech.
```

The TextToSpeech Kit is designed to be as flexible as possible, and to not limit the number and variety of uses that the programmer may envisage for the system. How the TextToSpeech Object will be used depends in large part on the nature of the application that the developer is building. In spite of this variety, however, the programmer will, in general, observe the following conventions in order when programming with the TextToSpeech Kit:

1. Instantiate and initialize the TextToSpeech Object using the **alloc** and **init** methods. This establishes a connection to the Server.

2. Configure the TextToSpeech Object so that it can do what the application requires of it. This may involve setting such things as voice quality and dictionary order.

3. Use the text input and real-time methods to produce speech as desired.

4. Free the TextToSpeech Object from memory using the **free method.** This also severs the connection to the Server.

3.2 A Simple Example

In this section we will develop a simple application which allows the user to enter text into a text field, and then hear it with the push of a button. Step-by-step instructions are given, so you should be able to follow along and create the application at your computer.

1. Create a subdirectory to work in. Give it a name such as "MySpeaker."

2. Start up Project Builder.

3. Click on the "New..." submenu item under the "Project" main menu item. A "New Project" panel will appear. Click on "MySpeaker" in the browser, and then push "OK". This creates the MySpeaker.nib interface in the English.lproj subdirectory, plus other files needed to manage the project.

4. Next click on the Libraries item in the Files browser in Project Builder. Double click on the Libraries suitcase. An "Add Libraries" panel will appear. Add the library `libTextToSpeech.a` located in /usr/local/lib.

5. Save the project by selecting the "Save" menu item.

6. Start up Interface Builder by double clicking on MySpeaker.nib in the Files browser of Project Builder (under Interfaces).

7. Drag a text field from the Palette and place it on "My Window." Resize and position it to taste. You should delete the word "Text" from the field.

8. Drag a button from the Palette and put it in the same window. Resize and position it to taste. Rename the button "speak." If desired, you can also resize and rename the window.

9. Next you need to create a class to control the actions of your application. To do this, click on the "Classes" suitcase in the Interface Builder window in the lower left corner of the screen. Click on the left-arrow of the browser until "Object" appears in the left-hand column. Then click once on "Object" so that nothing is highlighted in the right-hand column. Drag the pull-down "Operations" menu until the mouse is positioned over "Subclass," and then release the mouse button. This creates a subclass of Object called "MyObject." Rename this to "Controller" in the Class Inspector.

10. In the Class Inspector, add an outlet called "myText," and an action called "speak."

11. Select the "Unparse" item of the Operations pull-down menu. Interface Builder will ask you if you wish to create Controller.h and Controller.m, and if you want to add these files to your project. Answer "OK" and "Yes" to these questions.

12. Select the "Instantiate" item of the Operations pull-down menu. An icon with the name "Controller" will appear in the bottom window.

13. In this same window, control-drag from the icon named "File's Owner" to the newly created icon. The Inspector will change so that a connection can be made between these two objects. When you click on the "Connect" button, "Controller" is made the delegate of the application.

14. Control-drag from the "speak" button to the "Controller" icon. In the Inspector panel, select the "speak:" action in right-hand column of the browser, and then push the "Connect" button. Every time the "speak" button is pushed, a "speak" message will now be sent to "Controller."

15. Control-drag from "Controller" to the text field in the main window. Push the "Connect" button in the Inspector panel. This names the text field "myText."

16. Save the work you have done in Interface Builder by selecting the "Save" submenu item under the "File" menu item.

17. You are now ready to write some Objective C code. In an editor, open the file `Controller.m`. Add these two lines after the first "import" statement in the file:

```
#import <appkit/appkit.h>
#import <TextToSpeech/TextToSpeech.h>
```

18. In the same file, add this code just before the **speak:** method:

```
- appDidInit:sender
    {
    /*  CONNECT APPLICATION TO TEXT-TO-SPEECH SERVER  */
    mySpeaker = [[TextToSpeech alloc] init];
    if (mySpeaker == nil) {
        /*  TELL THE USER THAT NO CONNECTION CAN BE MADE  */
        NXRunAlertPanel("No Connection Possible",
           "Too many clients, or server cannot be started.",
           "OK", NULL, NULL);
        [NXApp terminate:self];
    }

    return self;
}

- appWillTerminate:sender
{
    /*  FREE UP A CLIENT SLOT  */
    [mySpeaker free];

    return self;
}
```

When the application is initialized, the **appDidInit** message is sent to the "Controller" object, since it has been made the delegate of the application. The **appDidInit: method connects the application to the TextToSpeech Server with the code:**

```
mySpeaker = [[TextToSpeech alloc] init];
```

The TextToSpeech Server is actually represented by the "mySpeaker" object, so any subsequent messages intended for the Server should be sent to this object. If the **init** message returns **nil**, this means that no connection can be made to the Server. In the code above, the application puts up a warning panel and terminates the application whenever this happens. The **appWillTerminate:** method is needed because an application should always sever its connection with the TextToSpeech Server when the application is about to quit. This frees up a client slot for other applications to use.

19. Add the following line to the **speak:** method in the `Controller.m` file:

```
[mySpeaker speakText:[myText stringValue]];
```

When the user pushes the "speak" button, the text will be retrieved from the text field and then sent to the TextToSpeech Server to be spoken.

20. In the `Controller.h` file, add the line:

```
id  mySpeaker;
```

    after the "myText" instance variable. The variable "mySpeaker" is where the id of the instantiated TextToSpeech Object is stored.

21. In the same file, add these lines just before the **speak:** method declaration:

```
- appDidInit:sender;
- appWillTerminate:sender;
```

22. Save `Controller.m` and `Controller.h` to disk.

23. Compile the application by clicking on the "Build" button under the Builder section of Project Builder. The file `MySpeaker.app` should be produced. If you encounter any compile errors, you will have to correct the source code using an editor.

24. Once the application has been successfully compiled, you can run it by double clicking on `MySpeaker.app` using Workspace Manager, or by clicking on the "Run" button in Project Builder. Type a word or two in the text field, and then push the "speak" button. You should hear the text spoken. If not, check your work, making sure the code is correct, and that all connections have been properly made.

Note that this application establishes a connection with the TextToSpeech Server when it is started up. A connection to the Server can be made at any time, but since it takes a second or two (especially if the Server itself needs to be started), it is usually better that the application does this before any real work needs to be done. Also note that the above application explicitly severs its connection with the Server by sending a **free** message. Although the Server can detect when a client application has terminated, it is bad form for an application to leave a "dangling" connection to the Server when it quits.

3.3 Adding More Control

In this section we add more control to the application developed above. First of all, we will add a slider to control the speed of the speech:

1. Using Project Builder and Interface Builder, open the project and nib files that were

developed in the section above.

2. Drag a slider from the Palette and place it in the main window of the application. Resize and position it to taste. Using the Attributes part of the Slider Inspector, set the minimum value to 0.2, the current value to 1.0, and the maximum value to 2.0.

3. Drag a text field from the Palette and place it near the slider. Delete the word "Text" from the field, and disable the "Editable" and "Selectable" options using the TextField Inspector. We will use this text field to display the value of the slider.

4. If desired, group the slider and text field together using a box, and title the box "Speed."

5. We need to add two outlets and one action to the Controller class. To do this, first click on the "Controller" class icon in the window in the bottom left-hand corner. Then click on the Classes suitcase icon. The Inspector should now change to show the Class Inspector for the Controller class. Now add the outlets "speedSlider" and "speedField,", and the action "speedSliderMoved."

6. Click on the Objects suitcase, and control-drag from the "Controller" icon to the slider. Set the connection to "speedSlider," and then push the "Connect" button.

7. Control-drag from the "Controller" icon to the text field. Set the connection to "speedField," and then push the "Connect" button.

8. Control-drag from the slider used to control the speed to the "Controller" icon. Set the connection to "speedSliderMoved:" and then push the "Connect" button.

9. Save your work in Interface Builder by selecting the "Save" menu item.

10. Open the `Controller.h` file with an editor

11. Add these two lines:

```
id  speedSlider;
id  speedField;
```

just after the other instance variables. These are the new outlets we added above.

12. In the same file, add the line:

```
- speedSliderMoved:sender;
```

just after the **speak:** method. This is the new action we added above.

13. Save the file `Controller.h` to disk.

14. Open the `Controller.m` file with an editor.

15. Add the following lines to the **appDidInit:** method, just before the return statement:

```
/*  SET SPEED SLIDER AND FIELD TO USER DEFAULTS  */ [speedSlider
setFloatValue:[mySpeaker speed]];
[speedField setFloatValue:[mySpeaker speed]];
```

    This sets the speed slider and text field to the speed the user has stored in the defaults database. When a TextToSpeech Object is instantiated (as has just been done in the **appDidInit:** method), this database is automatically used to set the defaults of the object. When our application is started up, the slider and text field will show the correct initial value of the object, because the speed has been has been found by sending the object the **speed** message.

16. Add the following method to the end of `Controller.m`:

```
- speedSliderMoved:sender
{
    /*  GET VALUE FROM SLIDER  */
    float value = [sender floatValue];

    /*  SET THE FIELD TO THIS VALUE  */
    [speedField setFloatValue:value];

    /*  IF MOUSE UP, SET THE VOICE TO THIS VALUE  */
    if ([NXApp currentEvent]->type == NX_LMOUSEUP)
       [mySpeaker setSpeed:value];

    return self;
}
```

    When the slider is moved, this method gets the current value of the slider and sets the field to that value. When the mouse has stopped dragging, the speed is then sent to the Server. This avoids unnecessary communication with the Server, and reduces the associated overhead.

17. Save the file `Controller.m` to disk.

18. Compile the application by building it in Project Builder. Correct any compiler errors.

19. Test the application by double clicking on “MySpeaker.app”, or by clicking on the

"Run" button in Project Builder. You should be able to vary the speed of the speech by moving the slider. If not, check your work.

Controls over volume, pitch, and stereo balance can be implemented in the same way. You may want to polish the application so that only one or two decimal digits are displayed in the text field, and so that a user can enter a speed value into the text field directly. An example of these improvements can be seen in the example source code for *BigMouth*.

Note that the slider, as implemented above, only sends a message to the TextToSpeech Server when the slider has stopped moving (i.e. when a mouse-up is detected). This avoids unnecessary communication with the Server, and reduces the overhead associated with inter-task communication.

Next we will add pause and continue controls to the application:

1. In Interface Builder, add the two actions "continue" and "pause" to the Controller class. Do this with the Class Inspector.

2. Drag two "item" menu selections from the Menu Palette, and place them in the main menu of the application. Rename one "pause", and the other "continue." If desired, you can also assign keyboard equivalents for each of these.

3. Control-drag from the "pause" menu item to the "Controller" icon. Select the "pause:" action in the MenuCell Inspector, and then push the "Connect" button.

4. Control-drag from the "continue" menu item to the "Controller" icon. Select the "continue:" action in the MenuCell Inspector, and then push the "Connect" button.

5. Save your work in Interface Builder.

6. Add these two method declarations to `Controller.h`:

```
- pause:sender;
- continue:sender;
```

7. Save the file `Controller.h` to disk.

8. Add these two methods to `Controller.m`:

```
- pause:sender
{
    [mySpeaker pauseImmediately];
    return self;
```

```
}

- continue:sender
{
    [mySpeaker continue];
    return self;
}
```

9. Save the file `Controller.m` to disk.

10. Build the application as before.

11. Test the application by entering a fairly long phrase. You should be able to pause the speech by clicking on the "pause" menu item, and resume it by clicking on the "continue" menu item.

Other real-time controls can be added in a similar manner. See the example code for *BigMouth* and *ServerTest*.

3.4 Customizing Pronunciations

In this section, we add the means to access custom pronunciations specially created for the application:

1. Using Interface Builder, open the application we developed above.

2. Using an editor, open the file `Controller.m`.

3. Add the following function after the last method in `Controller.m`:

```
void getAppDirectory (char *appDirectory)
{
    FILE *process;
    char command[256];
    char *suffix;

    strcpy (appDirectory,NXArgv[0]);
    if (appDirectory[0] == '/') {
        if (suffix = rindex(appDirectory,'/'))
        *suffix  = '\0';
    } else {
        sprintf(command,"which '%s'\n",NXArgv[0]);
        process=popen(command,"r");
        fscanf(process,"%s",appDirectory);
        pclose(process);
```

```
        if (suffix = rindex(appDirectory,'/'))
            *suffix  = '\0';
        chdir(appDirectory);
        getwd(appDirectory);
    }
}
```

This function, taken from *NeXTAnswers 642*, returns the full pathname of the directory from which the application has been launched.

4. Add the following declarations to the **appDidInit:** method:

```
char appDictPath[1024];
void getAppDirectory();
```

5. Add the following lines just before the return statement in the **appDidInit:** method:

```
/*  SET THE APPLICATION DICTIONARY  */ getAppDirectory(appDictPath);
strcat(appDictPath,"/appDictionary.preditor");
[mySpeaker setAppDictPath:appDictPath];
```

This sets the path to the Application Dictionary that will be placed in the application's file package. Note that, in this case, it must be named "appDictionary.preditor."

6. Save the file `Controller.m` to disk.

7. Build the application with "make install." This should create the application "MySpeaker.app" and install it in your /Apps directory.

8. Start the application *PrEditor* by double clicking its icon. *PrEditor* is installed in /LocalApps/TextToSpeech.

9. Create several custom pronunciations. Be sure to store the pronunciations. (Documentation for *PrEditor* is displayed by clicking on the "Help..." submenu item under "Info.")

10. Choose the "Save As..." submenu item under the "Document" menu item. The Save panel will appear. Save the document as "appDictionary.preditor" in the "MySpeaker.app" directory (i.e. put it in the file package for the MySpeaker application).

11. Test the application by double clicking on "MySpeaker.app." The application should pronounce the custom pronunciations you created with *PrEditor*. Check this by quitting the application, changing the pronunciations again (be sure to store and save), and relaunching the application.

Your Application Dictionary can contain as many custom pronunciations as you need. An Application Dictionary is very useful when an application will use specialized terminology, such as medical or foreign words, not found in the Main Dictionary.

3.5 Going Further

The above tutorials should provide a basic understanding of how to program using the TextToSpeech Kit. Of course, many refinements and enhancements are possible. Special modes of text entry can be added, as can ways of controlling pitch, intonation, stereo balance, and volume. Special text filters might be useful and can be linked into the system with the **speakStream:** method. User defaults can be utilized, and a different dictionary order might be appropriate for some applications.

The tutorials given above, plus the example code for *BigMouth*, *ServerTest*, and *TalkingCalculator* should serve a guide when you implement text-to-speech in your own applications. Just how you use the Kit really depends upon the nature of the application under development, so these examples may not be directly applicable. However, they should provide a starting point for further elaboration and experimentation.

Chapter 4

Class Summary

TextToSpeech

Inherits from:	Object
Declared in:	<TextToSpeech/TextToSpeech.h>
Library:	libTextToSpeech.a

4.1 Instance Variables

Inherited from Object	Class	isa;
Declared in TextToSpeech	int	SpeechIdentifier;
	float	serverVersionNumber;
	port_t	outPort;
	port_t	localPort;
	task_t	serverTaskPort;
	float	sampleRate;
	int	channels;
	float	balance;
	float	speed;
	int	intonation;
	int	voiceType;
	float	pitchOffset;
	float	vtlOffset;
	float	breathiness;
	float	volume;
	short	*dictionaryOrder;
	char	*appDictPath;
	char	*userDictPath;
	char	escapeCharacter;
	int	block;

```
int         softwareSynthesizer;
char        *version;
char        *dictVersion;
char        *tts_p;
char        *tts_lp;
char        *streamMem;
char        *localBuffer;
```

Note: These instance variables are private to the TextToSpeech class and cannot be directly changed by any subclass. Since these variables are copies of values in the Server, subclasses should set and get these values by sending the proper message to super, which either sets or retrieves the values in the TTS_Server task.

4.2 Method Types

4.2.1 Creating and Freeing the Object

+ **alloc**
+ **allocFromZone:**(NXZone *)*zone*
± **init**
± **free**

4.2.2 Voice Quality Methods

± (tts_error_t)**setOutputSampleRate:**(float)*rateValue*
± (float)**outputSampleRate**
± (tts_error_t)**setNumberChannels:**(int)*channelsValue*
± (int)**numberChannels**
± (tts_error_t)**setBalance:**(float)*balanceValue*
± (float)**balance**
± (tts_error_t)**setSpeed:**(float)*speedValue*
± (float)**speed**
± (tts_error_t)**setIntonation:**(int)*intonationMask*
± (int)**intonation**
± (tts_error_t)**setVoiceType:**(int)*voiceType*
± (int)**voiceType**
± (tts_error_t)s**etPitchOffset:**(float)*offsetValue*
± (float)**pitchOffset**
± (tts_error_t)**setVocalTractLengthOffset:**(float)*offsetValue*
± (float)**vocalTractLengthOffset**

± (tts_error_t)**setBreathiness:**(float)*breathinessValue*
± (float)**breathiness;**
± (tts_error_t)**setVolume:**(float)*volumeLevel*
± (float)**volume**

4.2.3 Dictionary Control Methods

± (tts_error_t)**setDictionaryOrder:**(const short *)*order*
± (const short *)**dictionaryOrder**
± (tts_error_t)**setAppDictPath:**(const char *)*path*
± (const char *)**appDictPath**
± (tts_error_t)**setUserDictPath:**(const char *)*path*
± (const char *)**userDictPath**

4.2.4 Text Input Methods

± (tts_error_t)**speakText:**(const char *)*text*
± (tts_error_t)**speakText:**(const char *)*text* **toFile:**(const char *)*path*
± (tts_error_t)**speakStream:**(NXStream *)*stream*
± (tts_error_t)**speakStream:**(NXStream *)*stream*
toFile:(const char *)*path*
± (tts_error_t)**setEscapeCharacter:**(char)*character*
± (char)**escapeCharacter**
± (tts_error_t)**setBlock:**(BOOL)*flag*
± (BOOL)**block**
± (tts_error_t)**setSoftwareSynthesizer:**(BOOL)*flag*
± (BOOL)**softwareSynthesizer**

4.2.5 Real-Time Methods

± (tts_error_t)**pauseImmediately**
± (tts_error_t)**pauseAfterCurrentUtterance**
± (tts_error_t)**continue**
± (tts_error_t)**eraseAllSound**
± (tts_error_t)**eraseCurrentUtterance**

4.2.6 Version Query Methods

± (const char *)**serverVersion**
± (const char *)**dictionaryVersion**

4.2.7 Error Reporting Methods

± (const char *)**errorMessage:**(tts_error_t)*errorNumber*

4.2.8 Archiving Methods

± **read:**(NXTypedStream *)*stream*
± **write:**(NXTypedStream *)*stream*
± **awake**

4.3 Constants and Defined Types

```
/*  Error return typedef  */
typedef int tts_error_t;


/*  Error Return Codes from the TextToSpeech Object  */
#define TTS_SERVER_HUNG                 (-2)
#define TTS_SERVER_RESTARTED            (-1)
#define TTS_OK                          0
#define TTS_OUT_OF_RANGE                1
#define TTS_SPEAK_QUEUE_FULL            2
#define TTS_PARSE_ERROR                 3
#define TTS_ALREADY_PAUSED              4
#define TTS_UTTERANCE_ERASED            5
#define TTS_NO_UTTERANCE                6
#define TTS_NO_FILE                     7
#define TTS_WARNING                     8
#define TTS_ILLEGAL_STREAM              9
#define TTS_INVALID_PATH                10
#define TTS_OBSOLETE_SERVER             11
#define TTS_DSP_TOO_SLOW                12
#define TTS_SAMPLE_RATE_TOO_LOW         13


/*  Output Sample Rate Definitions  */
#define TTS_SAMPLE_RATE_LOW             22050.0
#define TTS_SAMPLE_RATE_HIGH            44100.0
#define TTS_SAMPLE_RATE_DEF             22050.0


/*  Number Channels Definitions  */
#define TTS_CHANNELS_1                  1
#define TTS_CHANNELS_2                  2
#define TTS_CHANNELS_DEF                2
```

```
/*  Stereo Balance Definitions  */
#define TTS_BALANCE_MIN                 (-1.0)
#define TTS_BALANCE_MAX                 1.0
#define TTS_BALANCE_DEF                 0.0
#define TTS_BALANCE_LEFT                (-1.0)
#define TTS_BALANCE_RIGHT               1.0
#define TTS_BALANCE_CENTER              0.0


/*  Speed Control Definitions  */
#define TTS_SPEED_MIN                   0.2
#define TTS_SPEED_MAX                   2.0
#define TTS_SPEED_DEF                   1.0
#define TTS_SPEED_FAST                  1.5
#define TTS_SPEED_NORMAL                1.0
#define TTS_SPEED_SLOW                  0.5


/*  Intonation Definitions  */
#define TTS_INTONATION_DEF              0x1f
#define TTS_INTONATION_NONE             0x00
#define TTS_INTONATION_MICRO            0x01
#define TTS_INTONATION_MACRO            0x02
#define TTS_INTONATION_DECLIN           0x04
#define TTS_INTONATION_CREAK            0x08
#define TTS_INTONATION_RANDOMIZE        0x10
#define TTS_INTONATION_ALL              0x1f


/*  Voice Type Definitions  */
#define TTS_VOICE_TYPE_DEF              0
#define TTS_VOICE_TYPE_MALE             0
#define TTS_VOICE_TYPE_FEMALE           1
#define TTS_VOICE_TYPE_LARGE_CHILD      2
#define TTS_VOICE_TYPE_SMALL_CHILD      3
#define TTS_VOICE_TYPE_BABY             4


/*  Pitch Offset Definitions  */
#define TTS_PITCH_OFFSET_MIN            (-12.0)
#define TTS_PITCH_OFFSET_MAX            12.0
#define TTS_PITCH_OFFSET_DEF            0.0


/*  Vocal Tract Length Offset Definitions  */
#define TTS_VTL_OFFSET_MIN              (-3.0)
#define TTS_VTL_OFFSET_MAX              3.0
#define TTS_VTL_OFFSET_DEF              0.0
```

```
/*  Breathiness Definitions  */
#define TTS_BREATHINESS_MIN             0.0
#define TTS_BREATHINESS_MAX             10.0
#define TTS_BREATHINESS_DEF             0.5


/*  Volume Level Definitions  */
#define TTS_VOLUME_MIN                  0.0
#define TTS_VOLUME_MAX                  60.0
#define TTS_VOLUME_DEF                  60.0
#define TTS_VOLUME_LOUD                 60.0
#define TTS_VOLUME_MEDIUM               54.0
#define TTS_VOLUME_SOFT                 48.0
#define TTS_VOLUME_OFF                  0.0


/*  Dictionary Ordering Definitions  */
#define TTS_EMPTY                       0
#define TTS_NUMBER_PARSER               1
#define TTS_USER_DICTIONARY             2
#define TTS_APPLICATION_DICTIONARY      3
#define TTS_MAIN_DICTIONARY             4
#define TTS_LETTER_TO_SOUND             5


/*  Escape Character Definition  */
#define TTS_ESCAPE_CHARACTER_DEF        0x1B


/*  TTS NXDefaults Definitions  */
#define TTS_NXDEFAULT_OWNER             “TextToSpeech”
#define TTS_NXDEFAULT_SAMPLE_RATE       “sampleRate”
#define TTS_NXDEFAULT_CHANNELS          “channels”
#define TTS_NXDEFAULT_BALANCE           “balance”
#define TTS_NXDEFAULT_SPEED             “speed”
#define TTS_NXDEFAULT_INTONATION        “intonation”
#define TTS_NXDEFAULT_VOICE_TYPE        “voiceType”
#define TTS_NXDEFAULT_PITCH_OFFSET      “pitchOffset”
#define TTS_NXDEFAULT_VTL_OFFSET        “vtlOffset”
#define TTS_NXDEFAULT_BREATHINESS       “breathiness”
#define TTS_NXDEFAULT_VOLUME            “volume”
#define TTS_NXDEFAULT_USER_DICT_PATH    “userDictPath”


/*  TTS Defines for server access  */
#define TTS_NXDEFAULT_ROOT_USER         “root”
#define TTS_NXDEFAULT_SYSTEM_PATH       “systemPath”
#define TTS_SERVER_NAME                 “TTS_Server”
#define TTS_CLIENT_SLOTS_MAX            50
```

Chapter 5

Class Methods

5.1 Creating and Freeing the Object

alloc

+ **alloc**

Allocates memory in the default zone for the TextToSpeech Object. The object must then be initialized immediately using the **init** method. For example:

```
mySpeaker = [[TextToSpeech alloc] init];
```

See also: **allocFromZone:, free, init**

allocFromZone:

+ **allocFromZone:**(NXZone *)*zone*

Allocates memory in the specified *zone* for the TextToSpeech Object. The object must then be initialized immediately using the **init** method. For example:

```
mySpeaker = [[TextToSpeech allocFromZone:myZone] init];
```

Of course, the programmer can use the inherited method **zone** to specify the zone of another object.

See also: **alloc, init, free**

init

± init

Initializes the TextToSpeech Object. This method must be invoked immediately after the object has been allocated memory. If the TextToSpeech server cannot be started, or if all client slots are occupied, then this method will return **nil**. It is best to check for this immediately after allocating and initializing the object, as in the following example:

```
mySpeaker = [[TextToSpeech alloc] init];
if (mySpeaker == nil) {
    /*  TELL THE USER THAT NO CONNECTION CAN BE MADE  */
    NXRunAlertPanel("No Connection Possible",
       "Too many clients, or server cannot be started.",
       "OK", NULL, NULL);
    [NXApp terminate:self];
}
```

If **init** returns **nil**, there is no need to free the allocated object, since deallocation is handled automatically in this case.

The server is automatically launched when the client instantiates the Text-to-Speech Object (unless it has already been launched by another client application), and is freed whenever no clients are connected. Instantiating the Object creates a connection between the client application and the server; a single client slot in the server space thus becomes occupied. An application can instantiate as many Text-to-Speech Objects as needed (within system limits). This is the same, in effect, as having multiple connections to the server. Currently, the server can handle a maximum of 50 clients.

See also: **alloc, allocFromZone:, free**

free

± free

Frees the TextToSpeech Object in memory, and returns **nil**. Freeing the Object causes the connection between the client and server to be dropped, and frees a client slot in the server.

See also: **alloc, allocFromZone:**

5.2 Voice Quality Methods

setOutputSampleRate:

± (tts_error_t)**setOutputSampleRate:**(float)*rateValue*

Sets the output sample rate of the speaking voice. The argument *rateValue* must be set to either 44100 or 22050 Hz, or the `TTS_OUT_OF_RANGE` error is returned and the rate is set to the nearest in-range value. The following constants can also be used:

- TTS_SAMPLE_RATE_LOW
- TTS_SAMPLE_RATE_HIGH
- TTS_SAMPLE_RATE_DEF

The default rate is 22050 Hz. Due to limits in DSP hardware speed, the higher rate is only available when speaking to file. If you attempt to synthesize text to real-time output at the higher rate (using **speakText:** or **speakStream:**), no speech is synthesized and the `TTS_DSP_TOO_SLOW` error is returned.

The **setOutputSampleRate:** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if the argument is out of range, and `TTS_OK` otherwise. Out-of-range values for *rateValue* are reset to the nearest in-range value.

When using the DSP to synthesize shorter vocal tract lengths (under 15.9 cm), the higher sample rate must be used. Effectively, this means that voices that have short vocal tracts (children and smaller women) can only be synthesized to file with the present hardware. If you synthesize speech to file using the low sample rate and a vocal tract length under 15.9 cm, the file is created using the higher sample rate and the `TTS_SAMPLE_RATE_TOO_LOW` error is returned.

If you use the software synthesizer to synthesize to file, either the high or low sample rate can be used. Unlike the DSP synthesizer, the software synthesizer is far too slow to work in real time.

See also: **outputSampleRate, setVoiceType:, setSoftwareSynthesizer:, setVocalTractLengthOffset:**

outputSampleRate

± (float)**outputSampleRate**

Returns the current output sample rate. The returned value is a float

set to either 22050 or 44100 Hz.

See also: **setOutputSampleRate:**

setNumberChannels:

± (tts_error_t)**setNumberChannels:**(int)*channelsValue*

Sets the number of channels used for sound output. The argument *channelsValue is an integer that must be set to 1 or 2. The default number of channels is 2. The following constants can also be used:*

- *TTS_CHANNELS_1*
- *TTS_CHANNELS_2*
- *TTS_CHANNELS_DEF*

The **setNumberChannels:** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if the argument is out of range, and `TTS_OK` otherwise. Out-of-range values for *channelsValue* are reset to the nearest in-range value.

You can't adjust the stereo balance unless the number of channels is set to 2. When synthesizing to real-time output, using 1 channel is effectively the same as using 2 channels and setting the stereo balance to 0.0 (center position). This is because you cannot disable a single channel of the stereo D/A converter. Note, however, that when synthesizing to file, half as many samples are created when 1 channel is used. Thus, you normally only use a mono setting to save space when speaking to file.

See also: **numberChannels, setBalance:, balance**

numberChannels

± (int)**numberChannels**

Returns the current number of channels used for sound output. An integer with the value 1 or 2 is returned.

See also: **setNumberChannels:, setBalance:, balance**

setBalance:

± (tts_error_t)**setBalance:**(float)*balanceValue*

Sets the stereo balance of the speaking voice. The argument *balanceValue is a float that ranges from ±1.0 (hard left) to +1.0 (hard right). The default is 0.0 (center). The following constants are also available:*

- *TTS_BALANCE_LEFT*
- *TTS_BALANCE_RIGHT*
- *TTS_BALANCE_CENTER*

The **setBalance:** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if the argument is out of range, and `TTS_OK` otherwise. Out-of-range values for *balanceValue* are reset to the nearest in-range value. Any change of stereo balance is heard only when the number of output channels has been set to 2.

See also: **balance, setNumberChannels:, numberChannels**

balance

± (float)**balance**

Returns the current stereo balance. The returned value is a float which will range from ±1.0 (hard left) to +1.0 (hard right).

See also: **setBalance:, setNumberChannels:, numberChannels**

setSpeed:

± (tts_error_t)**setSpeed:**(float)*speedValue*

Sets the speed of the voice. The argument *speedValue* must be a float within the range 0.2 to 2.0. The value 1.0 is the normal rate of speech. The speed is varied by setting *speedValue* to a factor of this normal rate. For example, the value 2.0 indicates that the speech will be twice the normal rate, and the value 0.5 indicates that the speech will be one half the normal rate. The *speedValue argument can also be set using the definitions contained in TextToSpeech.h. The available choices are:*

- *TTS_SPEED_FAST,*

- *TTS_SPEED_NORMAL,*
- *TTS_SPEED_SLOW.*

The **setSpeed** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if the argument is out of range, and `TTS_OK` otherwise. Out-of-range values for *speedValue* are reset to the nearest in-range value.

See also: **speed**

speed

± (float)**speed**

Returns the current speed of the TextToSpeech Object. The returned value is a float within the range of 0.2 to 2.0.

See also: **setSpeed:**

setIntonation:

± (tts_error_t)**setIntonation:**(int)*intonationMask*

Sets the type and extent of intonation used by the synthesized voice. Intonation is set by bitwise OR'ing the desired options listed below:

- *TTS_INTONATION_NONE*
- *TTS_INTONATION_MICRO*
- *TTS_INTONATION_MACRO*
- *TTS_INTONATION_DECLIN*
- *TTS_INTONATION_CREAK*
- *TTS_INTONATION_RANDOMIZE*
- *TTS_INTONATION_ALL*

Micro-intonation is variation in pitch due to segmental level changes in air pressure associated with events such as stops and plosives; its effect is quite subtle. Macro-intonation results from supra-segmental features such as syllable stress, phrase structure, sentence type, etc. Declination (`TTS_INTONATION_DECLIN`) is the gradual drop in pitch that naturally occurs in most sentences. Creaky voice (`TTS_INTONATION_CREAK`) results when the voice drops in pitch to such an extent that a pulse-like quality is heard (currently not implemented). When the randomize bit is set, slight pitch variations are added to the macro-intonation.

If no intonation at all is desired, then *intonationMask* should be set to TTS_INTONATION_NONE. If intonation from all sources is desired, then set the argument to TTS_INTONATION_ALL. This is the default setting.

The **setIntonation:** method takes effect only after the current sentence has finished speaking. The method returns TTS_OUT_OF_RANGE if the argument is out of range, and TTS_OK otherwise. Out-of-range values for *intonationMask* are reset to the default.

See also: **intonation**

intonation

± (int)**intonation**

Returns the type of intonation currently in use. The returned value can be decoded by bitwise AND'ing it with the constants listed in **setIntonation:** above.

See also: **setIntonation:**

setVoiceType:

± (tts_error_t)**setVoiceType:**(int)*voiceType*

Sets the voice type of the speaking voice. The global definitions contained in TextToSpeech.h may be used to set *voiceType. Currently, the available choices are:*

- *TTS_VOICE_TYPE_MALE,*
- *TTS_VOICE_TYPE_FEMALE,*
- *TTS_VOICE_TYPE_LARGE_CHILD*
- *TTS_VOICE_TYPE_SMALL_CHILD*
- *TTS_VOICE_TYPE_BABY*

TTS_VOICE_TYPE_MALE is the default. The **setVoiceType:** method takes effect only after the current sentence has finished speaking. The method returns TTS_OUT_OF_RANGE if the argument is out of range, and TTS_OK otherwise. Out-of-range values for *voiceType* are reset to the default.

Setting the voice type sets the baseline values for the median pitch and vocal tract length. You can "offset" from these baseline values by calling the **setPitchOffset:** or **setVocalTra**

ctLengthOffset: methods. These offsets remain in effect when you change voice type. This means, for example, that if you offset the pitch to create a low male voice, when you switch to a female voice, it too will be a "low" voice.

The baseline values for each voice type are summarized below:

	Male	*Female*	*Large Child*	*Small Child*	*Baby*
length	*17.5*	*15.0*	*12.5*	*10.0*	*7.5*
pitch	*±12.0*	*0.0*	*2.5*	*5.0*	*7.5*

The vocal tract length is given in centimeters, and the pitch is given in semitones, with 0 equal to middle C.

Note that smaller vocal tract lengths cannot be synthesized in real time on the DSP (this varies from platform to platform, and from sound card to sound card). Also note that there are other restrictions when synthesizing to file using the DSP. See the Concepts chapter for details.

See also: **voiceType, setPitchOffset:, setVocalTractLengthOffset:**

voiceType

± (int)**voiceType**

Returns an integer which indicates what voice type is currently in use. The integer will correspond to one of the definitions contained in TextToSpeech.h. The available choices are:

- **TTS_VOICE_TYPE_MALE,**
- **TTS_VOICE_TYPE_FEMALE,**
- **TTS_VOICE_TYPE_LARGE_CHILD**
- **TTS_VOICE_TYPE_SMALL_CHILD**
- **TTS_VOICE_TYPE_BABY**

See also: setVoiceType:

setPitchOffset:

± (tts_error_t)**setPitchOffset:**(float)*offsetValue*

Sets the pitch offset of the speaking voice, in semitones. This method is used to raise or

lower the “center pitch” of the speaking voice. Naturally, each voice type will have its own unique center pitchÐthis method allows one to customize a voice type. For example, when the voice type is set to `TTS_VOICE_TYPE_MALE`, a deep baritone could be specified by offsetting the pitch downwards by several semitones. Of course, if intonation is enabled, the pitch of the voice will vary around its center pitch.

The argument *offsetValue* is a float with an allowable range of ±12.0 semitones (one octave up or down). Note that fractional semitones are possible. The **setPitchOffset:** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if the argument is out of range, and `TTS_OK` otherwise. Out-of-range values for *offsetValue* are reset to the nearest in-range value. The default is 0.0.

See also: **pitchOffset, setVoiceType:**

pitchOffset

± (float)**pitchOffset**

Returns the current pitch offset, in semitones. A float within the range of ±12.0 (one octave up or down) is returned.

See also: **setPitchOffset:**

setVocalTractLengthOffset:

± (tts_error_t)**setVocalTractLengthOffset:**(float)*offsetValue*

Sets the vocal tract length offset of the speaking voice. This method is used to add to or subtract from the baseline vocal tract length associated with each voice type. For example, if a male voice has been specified, the vocal tract length is normally 17.5 cm. The vocal tract can be lengthened to 18.0 cm by setting the offset to 0.5 cm. If the voice type is then changed to female, the new vocal tract length will be 15.5 cm, since the 0.5 cm offset is added to baseline length of 15.0 cm.

The argument *offsetValue* is a float with an allowable range of ±3.0 centimeters. The **setV ocalTractLengthOffset:** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if the argument is out of range, and `TTS_OK` otherwise. Out-of-range values for *offsetValue* are reset to the nearest in-range value. The default is 0.0.

See also: **vocalTractLengthOffset, setVoiceType:**

vocalTractLengthOffset

± (float)**vocalTractLengthOffset**

Returns the current vocal tract length offset, in centimeters. A float within the range of ±3.0 is returned.

See also: **setVocalTractLengthOffset:**

setBreathiness:

± (tts_error_t)**setBreathiness:**(float)*breathinessValue*

Sets the breathiness of the synthesized speaking voice. The argument *breathinessValue* is a float with an allowable range of 0.0 to 10.0. 0.0 means no breathiness, and larger numbers mean that more noise is added to the glottal source waveform.

The **setBreathiness:** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if the argument is out of range, and `TTS_OK` otherwise. Out-of-range values for *breathinessValue* are reset to the nearest in-range value. The default value is 0.5.

See also: **breathiness**

breathiness

± (float)breathiness

Returns the current breathiness value. A float within the range of 0.0 to 10.0 is returned.

See also: **setBreathiness:**

setVolume:

± (tts_error_t)**setVolume:**(float)*volumeLevel*

Sets the volume of the synthesized speaking voice. The argument *volumeLevel* is a float with an allowable range of 0.0 to 60.0. The *volumeLevel is in decibels, with 60.0 dB the maximum, and 0.0 dB silence. The following constants are also available:*

- *TTS_VOLUME_LOUD,*
- *TTS_VOLUME_MEDIUM,*
- *TTS_VOLUME_SOFT,*
- *TTS_VOLUME_OFF.*

TTS_VOLUME_LOUD (60.0 dB) is the default value. The **setVolume:** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if the argument is out of range, and `TTS_OK` otherwise. Out-of-range values for *volumeLevel* are reset to the nearest in-range value.

See also: **volume**

volume

± (float)**volume**

Returns the current volume level of the speaking voice in decibels. The returned value is a float which will range from a minimum of 0.0 dB to a maximum of 60.0 dB.

See also: **setVolume:**

5.3 Dictionary Control Methods

setDictionaryOrder:

± (int)**setDictionaryOrder:**(const short *)*order*

Sets the dictionary search order. This gives the user a high degree of control over which dictionaries are used for word pronunciations, and in what order the dictionaries are searched. The desired dictionaries are placed in a list, numbered from 1 to *n.* When the pronunciation for a word is needed, the dictionaries are searched in order until the pronunciation is found. The pronunciation is always taken from the first dictionary in which it is found; dictionaries further down in order are not searched. There are five dictionaries available:

1. Number parser. Actually an algorithm. Handles any string which contains a digit,

including all cardinal and ordinal numbers, decimal numbers, fractions of the form `a/b`, positive and negative quantities, percentages, clock times of the form `[d]d:dd[:dd]`, phone numbers in North American format, among others.

2. User dictionary. This dictionary contains user-specified pronunciations, and is local to each user's account. This file is built and maintained with the utility called *PrEditor*.

3. Application dictionary. This dictionary contains developer-specified pronunciations, and is local to the application. This file is built and maintained with the utility called *PrEditor.*

4. Main dictionary. This dictionary is the main source for pronunciations. It contains pronunciations for about 100,000 words.

5. Letter-to-sound. Actually an algorithm. It is designed to catch any words not found in any of the other dictionaries since it provides relatively crude pronunciations based on letter-to-sound rules.

The argument order is a pointer to a four-element array declared by the user. The search order is set by initializing this array with the following constants:

- *TTS_NUMBER_PARSER*
- *TTS_USER_DICTIONARY*
- *TTS_APPLICATION_DICTIONARY*
- *TTS_MAIN_DICTIONARY*
- *TTS_LETTER_TO_SOUND*
- *TTS_EMPTY*

For example, the default search order is: number parser, user dictionary, application dictionary, main dictionary, and letter-to-sound. This could be explicitly set with:

```
short int order[4] = {TTS_NUMBER_PARSER,
                      TTS_USER_DICTIONARY,
                      TTS_APPLICATION_DICTIONARY,
                      TTS_MAIN_DICTIONARY};
[mySpeaker setDictionaryOrder:order];
```

Note that `TTS_LETTER_TO_SOUND` is not specified in this list. This is because the letter-to-sound algorithm cannot be removed from the search order since it is a "catch-all" designed to provide a pronunciation if none can be found in any of the other dictionaries. If the letter-to-sound algorithm is put before another dictionary, then that dictionary will never be referenced since the letter-to-sound algorithm is guaranteed to find a pronunciation of some

sort.

Any dictionary or algorithm except the letter-to-sound algorithm can be left out of the search order. For example, if one wants to leave out the user dictionary and number parser, but retain the main dictionary, application dictionary, and letter-to-sound algorithm, the following would be used:

```
short int order[4] = {TTS_MAIN_DICTIONARY,
                      TTS_APPLICATION_DICTIONARY,
                      TTS_LETTER_TO_SOUND,
                      TTS_EMPTY};
[mySpeaker setDictionaryOrder:order];
```

In this case the letter-to-sound algorithm is explicitly set to be third in the search order, but this is somewhat redundant since the letter-to-sound algorithm can never be removed from the search order. The above example is equivalent to:

```
short int order[4] = {TTS_MAIN_DICTIONARY,
                      TTS_APPLICATION_DICTIONARY,
                      TTS_EMPTY,
                      TTS_EMPTY};
[mySpeaker setDictionaryOrder:order];
```

Note that `TTS_EMPTY` must be used for any unused entries, and that unused entries must always be at the end of the order list. If `TTS_EMPTY` occurs at the beginning or in the middle of the list, then the method returns an error code, and resets the order to the default. Repeating any item except `TTS_EMPTY` in the order list also returns an error.

The **setDictionaryOrder** method takes effect only after the current sentence has finished speaking. The method returns `TTS_OUT_OF_RANGE` if any element of the *order* array is out of range or if any of the above input errors occur, and returns `TTS_OK` otherwise. The search order is reset to the default on these error conditions.

See also: **dictionaryOrder**

dictionaryOrder

± (const short *)**dictionaryOrder**

Returns the dictionary search order currently in use. The method returns a pointer to an array of short ints, each element of which corresponds to one of the constants defined in the **setDictionaryOrder:** method. This array contains four elements.

See also: **setDictionaryOrder:**

setAppDictPath:

± (int)**setAppDictPath:**(const char *)*path*

Sets the complete pathname of the file containing the Application Pronunciation Dictionary. The Application Dictionary is built and maintained by the utility *PrEditor*. This method returns `TTS_NO_FILE` if the file containing the dictionary does not exist or cannot be opened, and the pathname is set to a zero-length string. Otherwise, `TTS_OK` is returned.

See also: **appDictPath**

appDictPath

± (const char *)**appDictPath**

Returns a string which contains the current pathname of the Application Pronunciation Dictionary. A zero-length string is returned if the path is not set.

See also: **setAppDictPath:**

setUserDictPath:

± (int)**setUserDictPath:**(const char *)*path*

Sets the complete pathname of the file containing the User Pronunciation Dictionary. The User Dictionary is built and maintained by the utility *PrEditor*. This method returns `TTS_NO_FILE` if the file containing the dictionary does not exist or cannot be opened, and the pathname is set to a zero-length string. Otherwise, `TTS_OK` is returned.

Normally, this method will never be used in an application. Whenever a user dictionary is created (using *PrEditor*), an entry is created in the defaults database. When the Text-to-Speech Object is initialized, the path to the user dictionary is set to the path specified in the defaults database; thus, the application will normally never need to set the path explicitly. However, there may be cases where the programmer may wish to override the default pathÐthis method allows for such an eventuality.

See also: **userDictPath**

userDictPath

± (const char *)**userDictPath**

Returns a string which contains the current pathname of the User Pronunciation Dictionary. A zero-length string is returned if the path is not set.

See also: **setUserDictPath:**

5.4 Text Input Methods

speakText:

± (tts_error_t)**speakText:**(const char *)*text*

Enters text to be spoken into the real-time system. This method is non-blocking, unless the **setBlock:** method has been used to turn blocking on. The argument *text* is a pointer to a null-terminated ASCII string which contains the text. Each string is considered an *utterance*. Each utterance must contain one or more words (or if in letter mode, one or more symbols). There is no limit on the length of an utterance, although the programmer has more control over how the text is to be spoken if the text is divided into smaller units such as sentences, and a series of **speakText:**
calls are used. Remember that the text is referred to by a pointer, so the programmer must be sure not to destroy or overwrite the character string until the text has been spoken.

This method returns `TTS_PARSE_ERROR` if the input *text* cannot be successfully parsed. `TTS_SPEAK_QUEUE_FULL` is returned if too many utterances are queued up and haven't had a chance to be synthesized. `TTS_DSP_TOO_SLOW` is returned if the hardware is too slow to synthesize the text with the current vocal tract length and output sample rate. Otherwise, `TTS_OK` is returned.

The Text-to-Speech Object uses the DSP chip to synthesize the desired speech. If the DSP and/or sound-out hardware is in use when the **speakText:** message is sent, then the speech will not be heard. The DSP and sound-out hardware is reserved by the Text-to-Speech system *only* when the **speakText:** message is being processed. This means that the hardware is free at all other times, allowing it to be used for such things as system beeps.

The **speakText:** method attempts to produce the most natural sounding speech from the

information provided in the text. This means that punctuation such as commas, periods, colons, semi-colons, question marks, and exclamation marks are used by the system to group words into units, to insert pauses of appropriate length between units, and to provide the appropriate pitch contour for the utterance. Idiosyncratic use of punctuation marks thus may produce rather strange sounding speech.

The programmer can embed escape codes into the text to control how the text it to be spoken. These escape codes are always 3 characters long and use the following format:

ESC *modeSymbol modeMarker*

The **ESC** character (hexadecimal `1B`) indicates the beginning of an embedded escape code. It must always be followed by 2 characters. The *modeSymbol* is a character which indicates which mode is being invoked. The modes and their *modeSymbols* are:

Mode	**Symbol**
Letter	l or L
Emphasis	e or E
Silence	s or S

The *modeMarker* is a single character which indicates whether the mode is beginning or ending. The letters **b** or **B** indicate the beginning of the mode, while the letters **e** or **E** indicate the end of the mode. Escape codes should never be embedded in the middle of a word. The **ESC** character itself can be changed to another value by using the **setEscapeCharacter:** method.

Letter mode causes one or more words or symbols to be spelled out one character at a time, rather than pronounced as a word. For example, if one wanted the words “pear” and “pair” in the following sentence to be spelled out, the escape codes for letter mode would be embedded as follows:

```
Use the word ESC-LBpairESC-LE rather
than ESC-LBpearESC-LE.
```

Any amount of text can be put into letter mode. Remember that once you are in letter mode you must explicitly exit it using the ESC-le code. The letter mode escape codes should never be placed in the middle of a word, since a word must be either solely spelled or spoken.

Letter mode is useful for spelling out unpronounceable strings. Strings such as `/*` or `1:594A-3` are often found in program source files or data files, and are obviously unpronounceable as words. Such strings should be sent to the TextToSpeech Object in letter mode. Letter mode is also useful if you wish a number to be pronounced one

digit at a time, rather than as would be normally done. For example, `ESC-lb123ESC-le` would be pronounced as "one two three" instead of "one hundred and twenty-three". All this implies that the programmer may have to pre-parse the text to produce suitable pronunciations; every application is different in its needs.

Emphasis mode is used to emphasize one or more words in an utterance. Usually it is applied to a single word which normally does not receive emphasis in a sentence. By placing the stress on that word the meaning of the utterance is altered. For example, by putting emphasis on the word "was" in the following sentence, it indicates that what was once true is no longer true, a considerable change from the sentence's normal declarative meaning:

```
He ESC-eb was ESC-ee an honest man.
```

Emphasis is often implied in text when a word is italicized or underlined. When spoken, an emphasized word is stressed, usually by making the word longer, louder, and higher in pitch. The Text-to-Speech system takes care of the emphasis, making sure it fits naturally with the rest of the words of the utterance. Note that the emphasis escape code cannot be placed in the middle of a word because only the whole word can be emphasized; partial emphasis of a word is not allowed.

Silence mode may be used by the programmer to insert silence between words. The amount of silence is specified in seconds or fractions of seconds with a decimal number. The specified length of silence will be rounded to the nearest 10th of a second, since that is the resolution of control. One must remember that silence is automatically inserted between words wherever a pause will naturally occur, such as after a period or comma. Silence mode should not be used to replace such naturally occurring pauses, but rather to specify arbitrary lengths of silence between words or utterances. If silence mode is used where there would normally be a pause, then that pause is discarded and the length of the pause becomes that which is specified with the embedded escape code. Silence mode cannot be used in the middle of a word; pauses are allowed between words only.

Silence is specified by embedding the escape code `ESC-sb` *time* between words. For example the text:

```
Come back to the Five and Dime, Jimmy Dean!
ESC-sb 3.0 Jimmy Dean!
```

has a 3 second silence inserted between the repetition of the words "Jimmy Dean". Normally there would be a short pause after the exclamation mark, but in this case the pause is arbitrarily specified by the programmer.

The `ESC-se` escape code is not necessary after the time, but can be inserted if desired to be consistent with letter and emphasis modes. Remember that the time values assigned using the silence mode are ASCII character representations of an integer or real number, not the integer or real data type itself. This is because the time value must be embedded within a character string.

See also: **setEscapeCharacter:, setBlock:**

speakText:toFile:

± (tts_error_t)**speakText:**(const char *)*text* **toFile:**(const char *)*path*

Converts *text* to speech and writes the resulting sound samples to the file with the specified *path* name. This method works exactly the same way as the **speakText:** method, except that output is directed to file instead of the sound output hardware; see the **speakText:** method for full details.

The output file can have 1 or 2 channels and either a 22050 or 44100 Hz sample rate (vocal tract lengths under 15.9 cm must use the 44100 sample rate when using the DSP to write to file). Its name should have a .snd suffix so that system commands such as *sndplay* and *sndinfo* will recognize the file. Note that the *sndconvert* command can be used on the resulting file to convert to another format or sampling rate. Also note that this method returns immediately, but the file will not be finished written to until all the text has been processed. Blocking can be turned on so that subsequent code will not be executed until the file is finished.

This method returns the same error codes as the **speakText:** method. Additionally, `TTS_INVALID_PATH` is returned if the argument *path* is a null pointer, points to a zero-length string, exceeds `MAXPATHLEN`, or specifies a file which cannot be opened for writing. `TTS_SAMPLE_RATE_TOO_LOW` is returned if the DSP is used to write to file using a vocal tract length under 15.9 cm at the low sample rate; the file is produced, but at the high sample rate.

See also: **speakText:, setEscapeCharacter:, setBlock:, setOutputSampleRate:, setNumberChannels:**

speakStream:

± (tts_error_t)**speakStream:**(NXStream *)*stream*

Enters all text contained in a memory stream into the real-time system. This method is non-blocking, unless the **setBlock:** method has been used to turn blocking on. The argument *stream* must point to an opened memory stream. The stream may be closed immediately after calling this method since the entire contents of the stream are copied to a local buffer. NeXT provides several C functions for reading and writing to streams, as well as mapping files to memory; see the *NEXTSTEP Reference Manual* for more details.

This method functions in exactly the same way as the **speakText:** method, except that the text to be spoken is stored in an memory stream. The stream need not be NULL terminated. The method returns the same error codes, plus `TTS_ILLEGAL_STREAM` if *stream* is invalid. This method only works with memory streams; all other stream types will not work and an error code will be returned. Note that many Application Kit streams are not memory streams (e.g. the stream in the Text class).

See also: **speakText:, setBlock:**

speakStream:toFile:

± (tts_error_t)**speakStream:**(NXStream *)*stream*
toFile:(const char *)*path*

Converts the text contained in the memory *stream* to speech and writes the resulting sound samples to the file with the specified *path* name. This method works exactly the same way as the **speakStream:** method, except that output is directed to file instead of the sound output hardware; see the **speakStream:** method for full details.

The output file can have 1 or 2 channels and either a 22050 or 44100 Hz sample rate (vocal tract lengths under 15.9 cm must use the 44100 sample rate when using the DSP to write to file). Its name should have a .snd suffix so that system commands such as *sndplay* and *sndinfo* will recognize the file. Note that the *sndconvert* command can be used on the resulting file to convert to another format or sampling rate. Also note that this method returns immediately, but the file will not be finished written to until all the text has been processed. Blocking can be turned on so that subsequent code will not be executed until the file is finished.

This method returns the same error codes as the **speakStream:** method. Additionally, `TTS_INVALID_PATH` is returned if the argument path is a null pointer, points to a zero-length string, exceeds `MAXPATHLEN`, or specifies a file which cannot be opened for writing. `TTS_SAMPLE_RATE_TOO_LOW` is returned if the DSP is used to write to file using a vocal tract length under 15.9 cm at the low sample rate; the file is produced, but at the high sample rate.

See also: **speakStream:, speakText:, speakText:toFile:, setBlock:**

setEscapeCharacter:

± **(tts_error_t)setEscapeCharacter:**(char)*character*

This method allows the programmer to change the value of the escape character for embedded escape codes used in the **speak...** methods. This may prove useful whenever the escape character must be visible, allowing easy editing of escape codes by hand. The default value of the escape character is hexadecimal `1B`.

This method returns `TTS_WARNING` if *character* is a printable character (including space). While such values are allowed for the escape character, they are not recommended since most texts will contain these characters as a matter of course. If the escape character is set to NULL or a non-ASCII value, then `TTS_OUT_OF_RANGE` is returned, and the escape character is reset to the default. `TTS_OK` is returned in all other cases.

See also: **escapeCharacter, speakText:**

escapeCharacter

± (char)**escapeCharacter**

Returns the value of the current escape character. The default value of the escape character is hexadecimal `1B`.

See also: **setEscapeCharacter:**

setBlock:

± **(tts_error_t)setBlock:**(BOOL)*flag*

Sets blocking off or on for the **speak...**methods. The use of blocking is strongly discouraged since no other processing can occur in your application until the entire text has been spoken. This means that the user will not be able to interrupt the synthesized speech with a pause method, or any of the erase methods. This is objectionable behavior for most applications, especially those that process long passages of text.

This method returns `TTS_OK` once the blocking has been set.

See also: **block**

block

± (BOOL)**block**

Returns a value which indicates if blocking has been set on or off for the **speak...** methods.

See also: **setBlock:**

setSoftwareSynthesizer:

± (tts_error_t)**setSoftwareSynthesizer:**(BOOL)*flag*

If set on, speech will be synthesized on the host CPU instead of the DSP when using the **speakText:toFile:**and **speakStream:toFile:** methods. This is useful if a DSP card is not available on Intel hardware. Note that the software synthesizer is extremely slow, and cannot be used for real-time synthesis. Also note that the software synthesizer will produce speech of somewhat better quality than the DSP, since a variable glottal pulse and high quality control-rate interpolation is used.

See also: **softwareSynthesizer**

softwareSynthesizer

± (BOOL)**softwareSynthesizer**

Returns a flag which indicates if the software synthesizer is used for the **speakText:toFile:** and **speakStream:toFile:** methods.

See also: **setSoftwareSynthesizer:**

5.5 Real-Time Methods

pauseImmediately

± (tts_error_t)**pauseImmediately**

Pauses the sound output immediately upon receipt of the method. If the system is already in a paused state, then the method is ignored, although an error code is returned. Pausing immediately may cause a "glitch" in the sound, especially if the pause occurs in the middle of a word or utterance. Sound output can be resumed by using the **continue** method. While in a paused state, new text can be entered into the system using the **speakText:** method, but will not be heard until the system is taken out of the paused state by using the **continue** method. Of course, any words left to be spoken at the time of the pause will be spoken before this new text can be heard. This can be overridden by using one of the "erase" methods described below.

This method returns `TTS_ALREADY_PAUSED` if the system is already in a paused state. `TTS_NO_UTTERANCE` is returned if no utterance exists to pause. Otherwise, `TTS_OK` is returned.

See also: **continue, eraseAllSound, eraseCurrentUtterance**

pauseAfterCurrentUtterance

± (tts_error_t)**pauseAfterCurrentUtterance**

Pauses the sound output after the currently speaking utterance upon receipt of the method. If the system is already in a paused state, then the method is ignored, although an error code is returned. Sound output at the start of the next utterance can be resumed by using the **continue** method. While in a paused state, new text can be entered into the system using the **speakText:** method, but will not be heard until the system is taken out of the paused state by using the **continue** method. Of course, any utterances left to be spoken at the time of the pause will be spoken before this new text can be heard. This can be overridden by using one of the "erase" methods described below.

This method returns `TTS_ALREADY_PAUSED` if the system is already in a paused state. `TTS_NO_UTTERANCE` is returned if no utterance exists in which to pause. Otherwise, `TTS_OK` is returned.

See also: **continue, eraseAllSound, eraseCurrentUtterance**

continue

± (tts_error_t)**continue**

Resumes sound output if the system is in a paused state, and returns `TTS_OK`. If the system

is not in a paused state, then the method is ignored, and TTS_NO_UTTERANCE is returned.

See also: **pauseImmediately, pauseAfterCurrentUtterance**

eraseAllSound

± (tts_error_t)**eraseAllSound**

Erases all sound left to be spoken upon receipt of the method, including any utterances after the current utterance. This method can be invoked in both the paused or unpaused state. All sound after the current speaking point or the pause point is erased. This will cause the latter portion of a word to be erased if the system is paused in the middle of a word (using the **pauseImmediately** method), or if the method is received while the system is still speaking a word. This method is useful for clearing the system after a pause, allowing fresh input to be immediately entered.

This method returns TTS_OK in all cases, even if there is no sound to erase.

See also: **eraseCurrentUtterance**

eraseCurrentUtterance

± (int)**eraseCurrentUtterance**

Erases all sound left to be spoken in the current utterance upon receipt of the method. This method can be invoked in both the paused or unpaused state. All sound after the current speaking point or the pause point to the end of the current utterance is erased. This will cause the latter portion of a word to be erased if the system is paused in the middle of a word (using the **pauseImmediately** method), or if the method is received while the system is still speaking a word. If the current utterance is paused, then the pause which is associated with this utterance is also erased, which means that if there are any other utterances queued up after this utterance, they will be heard immediately.

This method returns TTS_UTTERANCE_ERASED if the current utterance has already been erased. TTS_NO_UTTERANCE is returned if there is no utterance in which to erase. Otherwise, TTS_OK is returned.

See also: **eraseAllSound**

5.6 Version Query Methods

serverVersion

± (const char *)**serverVersion**

Returns a string which contains information about the current version of the Text-to-Speech Server.

See also: none

dictionaryVersion

± (const char *)**dictionaryVersion**

Returns a string which contains information about the current version of the Main Dictionary.

See also: none

5.7 Error Reporting Methods

errorMessage:

± (const char *)**errorMessage:**(tts_error_t)*errorNumber*

Returns a pointer to a printable error message. The argument *errorNumber* is the tts_error_t value returned by most methods of the TextToSpeech class.

See also: none

5.8 Archiving Methods

read:

± **read:**(NXTypedStream *)*stream*

Reads the TextToSpeech object from the typed stream *stream*. A **read:** message is sent in

response to archiving; you never send this message directly.

See also: **write:, awake**

write:

± **write:**(NXTypedStream *)*stream*

Writes the TextToSpeech object to the typed stream *stream*. A **write:** message is sent in response to archiving: you never send this message directly. Returns **self**.

See also: **read:, awake**

awake

± **awake**

Reinitializes the TextToSpeech object after it's been read in from a stream. This method ensures that a connection is made to the server.

An **awake** message is automatically sent to each object of an application after all objects of that application have been read in. You never send **awake** messages directly. The **awake** message gives the object a chance to complete any initialization that **read:** could not do. If you override this method in a subclass, the subclass should send this message to its superclass:

```
[super awake];
```

This method returns **self**, unless a connection cannot be made to the server. In this case, **nil** is returned.

See also: **read:, write:**